Broadview®
www.broadview.com.cn

Keras——一个极简的框架，帮您轻松入门深度学习

深度学习
Keras快速开发入门

乐毅 严超 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深度学习
Keras快速开发入门

乐毅 严超 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书首先介绍了 Keras 深度学习框架的技术背景、特点以及基本模型的构成，并比较了不同深度学习框架的优缺点。从 Keras 的安装、配置和编译等基本环境入手，详细介绍了 Keras 的模型、网络结构、数据预处理方法、参数配置，以及调试技巧和可视化工具。帮助读者快速掌握 Keras 深度学习框架，从而解决工作和学习当中神经网络模型的应用问题。同时，本书还介绍了如何用 Keras 快速构建深度学习原型并着手实战。最后通过 Cifar-10、词向量和对抗网络（GANs）等实例向读者展示 Keras 作为深度学习开发工具的强大之处，从而帮助读者迅速获得深度学习开发经验。

本书是一本实践性很强的深度学习工具书，适合希望快速学习和使用 Keras 深度学习框架的工程师、学者和从业者。本书对于立志从事深度学习和 AI 相关的行业，希望用 Keras 开发实际项目的工程技术人员，是非常实用的参考手册和工具书。

图书在版编目（CIP）数据

深度学习：Keras 快速开发入门 / 乐毅，严超编著. —北京：电子工业出版社，2017.8
ISBN 978-7-121-31868-9

Ⅰ. ①深… Ⅱ. ①乐… ②严… Ⅲ. ①软件设计 Ⅳ. ①TP311.5

中国版本图书馆 CIP 数据核字（2017）第 131017 号

责任编辑：孙学瑛
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：17.5　字数：178 千字
版　　次：2017 年 8 月第 1 版
印　　次：2017 年 8 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

1950 年，著名的图灵测试诞生，按照计算机科学和人工智能之父艾伦·麦席森·图灵（Alan Mathison Turing）的定义：“如果一台机器能够与人类展开对话，而不能被辨别出其机器身份，那么称这台机器具有智能。” 随后几年围绕着图灵测试模型产生了一系列的争论，直至 1956 年夏天，在美国达特茅斯大学召开的会议确立了“人工智能（Artificial Intelligence）”这一术语，并被认为是人工智能诞生的标志。时至今日，人工智能的发展经历了多次起伏，从 20 世纪 50 年代人工智能概念的兴起到 20 世纪 60 年代末到 20 世纪 70 年代的专家系统，再到 20 世纪 80 年代人工神经网络的飞速发展和 20 世纪 90 年代浅层机器学习的出现。2006 年加拿大多伦多大学教授、机器学习领域泰斗 Geoffrey Hinton 和他的学生 Ruslan Salakhutdinov 在顶尖学术刊物 Science 上发表了一篇文章“Reducing the Dimensionality of Data with Neural Networks”，开启了深度学习在学术界和工业界的浪潮。虽然人们对目前人工智能技术仍然存在许多争论，到底什么是真正的人工智能？是否存在像人一样能够独立思考的机器人？但这些疑问阻挡不了人类对人工智能的探索和向往。

2017 年 5 月，Google Deepmind 公司对去年发布的人工智能程序阿尔法围棋（AlphaGo）进行了优化和训练，并发起对战世界围棋冠军、职业九段选手柯洁的挑战，最终以 3：0 的总比分获胜，击破了代表人类智慧的最后的堡垒。

Deepmind 公司的围棋程序（AlphaGo）不同于传统的人工智能程序，它能够将大量棋谱数据输入计算机进行自我学习，而不像专家系统需要人类的智能去干预和调整算法。AlphaGo 的这一特点有力地证明了人工智能已经迎来了一个更高的发展阶段，即深度学习技术。

与此同时，国内外众多学者、从业者或业余爱好者都纷纷加入到深度学习（Deep Learning）的研究和工作中。然而，许多读者朋友对深度学习技术中复杂的数学公式和理论感到困惑，以至于常常无从下手。目前主流的深度学习框架，从 TensoFlow 难以理解的设计和表述方法到 Caffe 冗长的神经网络模型定义，往往使读者很难清晰、完整和快速地掌握深度学习技术。本书选择 Keras 这一深度学习框架向读者介绍深度学习技术和应用，力求使用简洁、高效和丰富的实例帮助读者快速掌握这门技术，这得益于 Keras 良好的模块化、极简的设计、快速原型迭代和易扩展性等特点。通过学习本书的内容，读者能够快速搭建满足产品需求的神经网络模型，从而加快产品研发周期。

本书共包括 9 章，每章的主要内容如下：

第 1 章“Keras 概述”介绍了 Keras 框架的技术背景、特点、优势以及与其他框架的对比。

第 2 章“Keras 的安装与配置”介绍了如何安装与配置 Keras，为进一步的 Keras 开发做准备。

第 3 章“Keras 快速上手”体现了 Keras 开发极易入手的特点，一定 Python 基础的开发人员都能非常简单地完成一个 Keras 模型训练。

第 4 章“Keras 模型的定义”详细介绍了 Keras 的两种类型模型——Sequential 模型和函数式模型，以及它们的参数定义和接口。

第 5 章“Keras 网络结构”深入 Keras 的内部结构，详细介绍了 Keras 的

网络结构及其层的定义，并对每层的参数进行说明和分析。

第 6 章“Keras 数据预处理”讲解了 Keras 提供的常用数据预处理工具方法，以及对于特定数据预处理使用场景的解析。

第 7 章“Keras 内置网络配置”介绍了 Keras 提供的内置网络配置，帮助读者完善模型优化体系的同时，熟悉 Keras 内置网络配置方法。

第 8 章“Keras 实用技巧和可视化”帮助读者完善了 Keras 实用技巧与可视化方法，打通 Keras 与下层框架（Theano、TensorFlow）的使用阻隔。

第 9 章“Keras 实战”完成了三个经典的 Keras 实战练习，增强读者对完整 Keras 训练过程的理解。

本书的内容由我与朋友严超共同完成，在写作过程中我们经常交流、探讨，互相指正对方的不足和错误，从而达到共同进步，在此对严超表示感谢。同时还要感谢电子工业出版社的孙学瑛老师对本书提出了很多非常好的建议，帮助我们顺利出版此书。

最后，由于深度学习技术发展迅速，各种知识和应用工具变化很快，Github 上 Keras 项目还处在活跃开发阶段，并即将发布支持 CNTK 和 MXNet 后端的实现，这使得 Keras 在不断更新和修正。笔者才疏学浅，理解有限，加之编写时间也较仓促，书中难免有错谬之处，敬请广大读者朋友批评指正，不胜感激。

乐　毅

2017 年 6 月

本书快览

目 录

第 1 章

Keras 概述

深度学习（Deep Learning）越来越成为人工智能领域研究和应用的热门方向，并不断涌现出很多优秀的深度学习框架。其中 Keras 框架自诞生以来，由于其良好的模块化，极简的设计和快速原型迭代等特性，越来越受到业内人士的青睐。也正由于这些特点，Keras 成为机器学习领域值得推荐的深度学习框架之一。

本章首先介绍了 Keras 的由来及技术背景，然后比较了当前主流的深度学习框架，最后总结了 Keras 框架的特点。

1.1 Keras 简介

1.1.1 Keras 2

Keras 深度学习框架的作者是 Francois Chollet，现就职于谷歌，从事机器学习与人工智能技术的研究。Keras 在 2015 年 3 月首次推出，现在用户数量已经突破了 10 万。其中有数百人为 Keras 代码库做出了贡献，更

有数千人为 Keras 社区做出了贡献。截止本书截稿之日，Keras 在 GitHub 上的 star 数已达 16333 个。Keras 不仅催生了众多创业公司，提高了研究者的成果率，而且简化了大公司的工程流程图，并为数以千计没有机器学习经验的工程师打开了一扇通向深度学习的大门。

今年 3 月，Keras 通过官方博客发布了全新的 Keras 2 版本，它带有一个更易使用的新 API，实现了与 TensorFlow 的直接整合。这是在 TensorFlow 核心整合 Keras API 所准备的重要一步。

Keras 2 相比以前的版本有很多新变化，主要体现在以下几个方面。

1. 与 TensorFlow 等后端框架整合

尽管 Keras 自 2015 年 12 月已经作为运行时后端（runtime backend）开始支持 TensorFlow，但 Keras API 却一直与 TensorFlow 代码库相分离。Keras 2 改变了这一情况，从 TensorFlow 1.2 版本开始，Keras API 可作为 TensorFlow 工作流的一部分直接使用，这是 TensorFlow 在向数百万新用户开源的道路上迈出的一大步。

Keras 作为一个 API 技术规范被理解和使用，而不是一个特殊的代码库。事实上，Keras 技术规范的发展会出现的两个不同分支：（a）TensorFlow 的内部实现（如 tf.keras），其完全由 TensorFlow 设计和实现，与 TensorFlow 的所有功能深度兼容；（b）外部的多后端实现，同时支持 Theano 和 TensorFlow（以后可能会支持更多的后台，如微软 CNTK 和 MXNET 后端已经处于密集开发阶段和发布前期）。

类似的，Skymind（商业智能和企业软件公司）正在用 Scala 语言实

现 Keras 的部分规范，如 ScalNet。为了在浏览器中运行，Keras.js 正在用 JavaScript 运行 Keras 的部分 API。正因如此，Keras API 将会成为深度学习从业者的通用语言，在不同的工作流程中共享并独立于底层平台。像 Keras 这样的统一 API 规范将促进代码共享，提高产品设计和实现的再生产率，从而进一步支持 Keras 的社区发展。

2. 全新 API

新的 Keras 2 为了提高重用性和向前兼容性，在这次发布中大量修改了 API。特别是新 API 选项完全兼容 TensorFlow 规范，具体变化如下[1]：

- 大多数神经网络层的 API 有了显著变化，特别是 Dense、BatchNormalization 和全卷积层。为了使 Keras 1 的代码在 Keras 2 上无障碍地运行，Keras 2 设置了兼容接口，代码运行的同时会有警告，提示迁移转换到新的 API 层接口。

- 训练和评估生成器方法的 API 发生了变化，比如 fit_generator、predict_generator 和 evaluate_generator 方法实现细节有一些更新，但这些 Keras 1 方法的调用依然适用于 Keras 2。

- 在 fit 中，训练 epoch 数的变量 nb_epoch 已重命名为 epochs。Keras 的 API 转换接口也适用于这项改变。

- 很多层所保存的权重格式已经改变。但是，Keras 1 上保存的权重文件依然能在 Keras 2 的模型上加载。

- Objectives 损失函数模块已更名为 Losses。

3. 显著修改

考虑到 Keras 的广大用户基础，Keras 2 并没有对 Keras 做根本变动。但有些变动不可避免，尤其是对于更高阶的用户，具体变化如下：

- 传统层 MaxoutDense、TimeDistributedDense 和 Highway 已被永久删除。
- 大量的传统度量和损失函数已被删除。
- BatchNormalization 层不再支持 mode 参数。
- Keras 内部构件已经改变，自定义层被升级。这些改变相对较小，只是使设计变得更快、更简单。
- 使用非正式的 Keras 功能编写的代码将会失效，因此高阶用户需要做一些相应的更新工作。

1.1.2 Keras 功能构成

1. Optimizers

顾名思义，Optimizers（优化器）包含了一些优化的方法，比如最基本的随机梯度下降 SGD。另外还有 Adagrad、Adadelta、RMSprop 和 Adam 等，一些新的方法以后也会被不断地添加进来。

```
keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.9,
nesterov=False)
```

上面的代码是 SGD 的使用方法，lr 表示学习速率；momentum 表示动量项；decay 是学习速率的衰减系数（每个 epoch 衰减一次）；nesterov 的

值是 False 或者 True，表示是否使用 nesterov momentum 动量方法。

2. Losses

Losses（损失函数）是计算神经网络输出与样本标记不一致程度的一种量化方法。Keras 提供了 mean_squared_error、mean_absolute_error、squared_hinge、hinge、binary_crossentropy、categorical_crossentropy 等常用的目标函数。这里 binary_crossentropy 和 categorical_crossentropy 也就是常说的 logloss（分别用作二类问题和多类问题）。

3. Activations

Activations（激活函数）的引入使得模型具有非线性因素，Keras 提供了 linear、sigmoid、hard_sigmoid、tanh、softplus、relu 和 softplus 等内置激活函数。另外，softmax 也放在 Activations 模块里。像 LeakyReLU 和 PReLU 这种比较新的激活函数，在 keras.layers.advanced_activations 模块里也可以找到。

4. Initializations

Initializations（参数初始化）是参数初始化模块，在添加 layer 的时候调用 init 进行初始化。Keras 提供了 uniform、lecun_uniform、normal、orthogonal、zero、glorot_normal 和 he_normal 等几种初始化方法。

5. Layer

Layer（层）模块包含了 core、convolutional、recurrent、advanced_activations、normalization 和 embeddings 等几种 layers。

其中，core 里面包含了 flatten（CNN 的全连接层之前需要把二维特征图展平成为一维的）、reshape（CNN 输入时将一维的向量变形成二维的）、dense（隐藏层，dense 是稠密的意思）等。比如，convolutional 层是对 Theano 的 Convolution2D 方法的封装。

6. Preprocessing

Preprocessing（预处理模块）包括序列数据的处理、文本数据的处理和图像数据的处理。在图像数据的处理中，Keras 提供了强大的 ImageDataGenerator 函数，实现 data augmentation（数据集增强），对图像做一些弹性变换，比如水平翻转、垂直翻转和旋转等。

7. Models

Models（模型）是最主要的模块，定义了各种基本组件，model 是将它们组合起来。

1.2 Keras 特点

Keras 作为基于 Theano 或 TensorFlow 的一个深度学习框架，它的设计参考了 Torch，使用纯 Python 语言编写，是一个高度模块化的神经网络库，支持 GPU 和 CPU。Keras 为支持快速实验而生，能够把想法迅速转换为结果。如图 1-1 所示，其特点如下：

（1）高度模块化。

（2）极简的语言。

（3）可扩展性。

（4）支持 CNN 和 RNN 或两者结合模型。

（5）无缝的 CPU 和 GPU 切换。

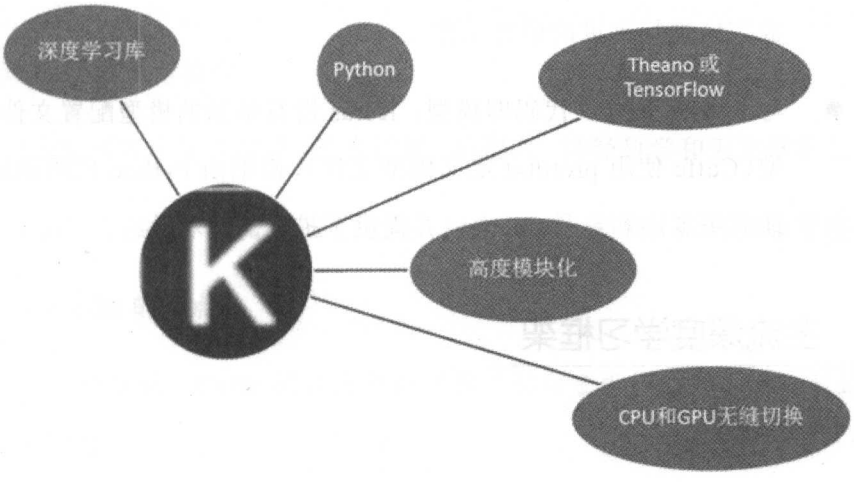


图 1-1　Keras 的特点

Keras 适用的 Python 版本是 2.7 及以上，目前最高达到 3.5 版本。

Keras 的设计原则有以下几个。

- 用户友好：Keras 是真正面对用户使用而设计的 API。用户的使用体验始终是 Keras 考虑的首要和中心内容。Keras 遵循减少认知困难的最佳实践，即提供一致而简洁的 API，能够极大地减少一般应用下用户的工作量，同时，Keras 提供清晰和具有实践意义的 bug 反馈。
- 模块性：模型可理解为一个层的序列或数据的运算图，完全可配置的模块可以用最少的代价自由组合在一起。具体而言，网络层、

损失函数、优化器、初始化策略、激活函数、正则化方法都是独立的模块，可以使用它们来构建自己的模型。

- 易扩展性：添加新模块非常容易，只需仿照现有的模块编写新的类或函数即可。创建新模块的便利性使得 Keras 更适合于“低门槛”地进行先进的研究工作。
- 与 Python 协作，代码即模型：Keras 没有单独的模型配置文件类型（Caffe 使用 prototxt 定义模型文件），模型由 Python 代码描述，使其更紧凑和更易 debug，并提供了扩展的便利性。

1.3 主流深度学习框架

1.3.1 Caffe

Caffe 是一个清晰而高效的深度学习框架。其作者是毕业于 UC Berkeley 的贾扬清博士，目前在 Google 工作。

Caffe 的全称是 Convolutional Architecture for Fast Feature Embedding，采用 BSD 2-Clause 开源协议，核心语言是 C++，支持命令行、Python 和 Matlab 接口，既可以在 CPU 上运行，也可以在 GPU 上运行。Caffe1 不支持分布式。

Caffe 的设计是建立在神经网络的一个简单假设，所有的计算都是以层的形式来表示的，网络中层做的事情就是输入数据，然后输出计算以后的结果。比如卷积，就是输入一个图像，随后和这一层的参数（filter）做卷积，最终输出卷积的结果。每层需要做两种函数计算：一种是 forward，

从输入计算到输出；另一种是 backward，从上层给的 gradient 来计算相对于输入层的 gradient。这两个函数实现了以后，我们就可以把很多层连接成一个网络。在这个网络输入数据（如图像、语音或其他原始数据），然后计算需要的输出（比如识别的标签）。在训练的时候，可以根据已有的标签计算 loss 和 gradient，然后用 gradient 来更新网络中的参数。这就是 Caffe 的基本工作流程。

Caffe 可以应用在视觉、语音识别、机器人、神经科学和天文学等领域。

Caffe 提供了一个完整的工具包，用来训练、测试、微调和部署模型。

1. Caffe 的特点

（1）模块化：Caffe 设计之初就做到了尽可能地模块化，允许对数据格式、网络层和损失函数进行扩展。

（2）表示和实现分离：Caffe 的模型（model）定义是用 Protocol Buffer 语言写进配置文件的。Caffe 支持网络架构，能根据网络的需要来正确占用内存，并且能够通过一个函数调用，实现 CPU 和 GPU 之间的切换。

（3）测试覆盖：在 Caffe 中，每一个单一的模块都对应一个测试。

（4）接口丰富：同时提供 Python 和 Matlab 接口。

（5）预训练参考模型：针对视觉项目，Caffe 提供了一些参考模型，这些模型只能应用在学术和非商业领域，它们的 License 不是 BSD。

2. Caffe 的架构

（1）数据存储：Caffe 通过“blobs”，即以四维数组的方式存储和传递

数据。blobs 提供了一个统一的内存接口，用于批量处理图像（或其他数据）操作和参数更新。模型（Model）以 Google Protocol Buffers 的方式存储在磁盘上。大型数据存储在 LevelDB 数据库中。

（2）层：一个 Caffe 层（Layer）是一个神经网络层的本质，它采用一个或多个 blobs 作为输入，并产生一个或多个 blobs 作为输出。网络作为一个整体的操作，层有两个关键职责，即前向传播，需要输入并产生输出；反向传播，取梯度作为输出，通过参数和输入计算梯度。Caffe 提供了一套完整的层类型。

（3）网络和运行方式：Caffe 保留了所有的有向无环层图，确保正确地进行前向传播和反向传播。Caffe 模型是终端到终端的机器学习系统。一个典型的网络开始于数据层，结束于 loss 层。通过一个单一的开关，使其网络运行在 CPU 或 GPU 上。在 CPU 或 GPU 上，层会产生相同的结果。

（4）训练一个网络：Caffe 训练一个模型（Model）靠快速、标准的随机梯度下降算法。在 Caffe 中，微调（fine tuning）是一个标准的方法，它适应于存在的模型、新的架构或数据。对于新任务，Caffe 微调旧的模型权重并按照需要初始化新的权重。

1.3.2 Torch

Torch 诞生已经有十多年，是一个广泛支持机器学习算法的科学计算框架，易于使用且高效，由简单和快速的脚本语言 LuaJIT 和底层 C/CUDA 实现。

Torch 的特点如下：

（1）具有强大的 n 维数组；

（2）具有丰富的索引、切片和 transposing 的例程。

（3）通过 LuaJIT 的 C 接口。

（4）线性代数例程。

（5）基于能量的神经网络模型。

（6）数值优化例程。

（7）支持快速、高效的 GPU。

（8）可移植嵌入到 iOS、Android 和 FGPA 平台。

Torch 目标是通过极其简单的过程、最大的灵活性和速度建立自己的科学算法。Torch 有一个大型生态社区驱动库包，包括计算机视觉软件包、信号处理、并行处理、图像、视频、音频和网络等，这些都基于 Lua 社区建立。

Torch 的核心是流行的神经网络，它使用简单的优化库，同时具有最大的灵活性，实现复杂的神经网络的拓扑结构。通过 CPU 和 GPU 等有效方式，可以建立神经网络和并行任意图。

Torch 广泛使用在学校的实验室，以及谷歌（DeepMind）、推特、NVIDIA、AMD、英特尔等公司里。

Facebook 开源了他们基于 Torch 的深度学习库包，这个版本包括 GPU 优化的大卷积网（ConvNets）模块，以及稀疏网络，这些通常被用在自然语言处理的应用中。ConvNet 模块包括 FFT-based 卷积层，使用的是建立

在 NVIDIA 的 CUFFT 库上自定义优化的 CUDA 内核。

1.3.3 Keras

Keras 是一个简约的、高度模块化的神经网络高层库，是基于 Theano 的一个深度学习框架。它的设计参考了 Torch，用 Python 语言编写，支持 GPU 和 CPU。其特点如下：

（1）使用简单，能够快速实现原型。

（2）支持卷积网络和递归网络，以及两者的组合。

（3）无缝运行在 CPU 和 GPU 上。

（4）支持任意连接方式，包括多输入/多输出训练。

Keras 库与其他采用 Theano 库的区别是 Keras 的编码风格非常简约、清晰。它把所有的元素操作使用小类封装起来，能够很容易地组合在一起，并创造出一种全新的模型。

1.3.4 MXNet

MXNet 是一个轻量化分布式可移植深度学习的计算平台。它支持多机多节点分布式以及多 GPU 的计算，其 openMP+MPI/SSH+Cuda/Cudnn 的框架的计算速度很快，且能够与分布式文件系统地结合，从而实现大数据的深度学习。MXNet 支持从单机到多 GPU、多集群的计算。其特点如下：

（1）基于赋值表达式建立计算图，类似于 TensorFlow、Theano、Torch 和 Caffe。

（2）支持内存管理，并对两个不交叉的变量重复使用同一内存空间。

（3）MXNet 的核心是使用 C++实现的，并提供了 C 风格的头文件，支持 Python、R、Julia、Go 和 JavaScript 等语言。

（4）支持 Torch。

（5）支持移动设备端发布。

1.3.5　TensorFlow

TensorFlow 是谷歌基于 DistBelief 进行研发的第二代人工智能学习系统，其命名来源于本身的运行原理。Tensor（张量）意味着 n 维数组，Flow（流）意味着基于数据流图的计算，TensorFlow 为张量从图像的一端流动到另一端的计算过程。TensorFlow 是将复杂的数据结构传输至人工智能神经网中进行分析和处理过程的系统。

TensorFlow 表达了高层次的机器学习计算，大幅简化了第一代系统，并且具备更好的灵活性和可延展性，可被用于语音识别或图像识别等多项机器深度学习领域。TensorFlow 对 2011 年开发的深度学习基础架构 DistBelief 进行了各方面的改进，它可在小到一部智能手机，大到数千台数据中心服务器的各种设备上运行。TensorFlow 完全开源，任何人都可以使用。TensorFlow 的特点如下：

（1）高度的灵活性。TensorFlow 不是一个严格的“神经网络”库，只要能够将计算表示为一个数据流图，就可以使用 TensorFlow。

（2）较强的可移植性。TensorFlow 既可以在 CPU 和 GPU 上运行，也

可以将模型在云端服务器或 Docker 容器里运行。

（3）自动求微分。TensorFlow 具有自动求微分的能力，只需定义预测模型的结构，将结构和目标函数结合，并添加数据，TensorFlow 就能自动计算相关的微分导数。

（4）多语言支持。TensorFlow 支持 C++、Python、Go、Java、Lua 和 JavaScript 等语言。

1.3.6 CNTK

CNTK（Computational Network Toolkit）是微软用于搭建深度神经网络的计算网络工具包，此项目已在 GitHub 上开源。CNTK 有一套强大的优化系统来训练和测试神经网络，它是以抽象的计算图形式构建的。

CNTK 支持两种方式来定义网络：一种是使用 Simple Network Builder，通过设置少量参数就能生成一个标准的神经网络；另一种是使用网络定义语言（Network Definition Language，NDL）。

CNTK 相比 Caffe、Theano、TensoFlow 等主流工具性能更强，灵活性更好，可扩展性更高。CNTK 支持 CNN、LSTM、RNN 等流行的网络结构，支持 CPU 和 GPU 模式。

CNTK 得到微软的支持，但目前 bug 较多，不太适合初学者学习。

1.3.7 Theano

Theano 是 BSD 许可证下发布的一个开源项目，是由 LISA（现 MILA）在加拿大魁北克的蒙特利尔大学开发的基于 Python 的深度学习框架，专

门用于定义、优化、求值数学表达式，其效率比较高，适用于多维数组。

Theano 的核心是一个 Python 的数学表达式的编译器。Theano 获取用户数据结构，使之成为一个使用 NumPy、高效本地库的非常高效的代码，并能在 CPU 或 GPU 上尽可能快地运行。Theano 的特点如下：

（1）紧密集成 NumPy。在 Theano 的编译函数中使用 numpy.ndarray。

（2）透明使用 GPU。使得执行数据密集型的计算速度高达使用 CPU 的 140 倍（仅对浮点数操作）。

（3）高效的符号分解。Theano 计算一个或多个输入函数的推导。

（4）速度和稳定性优化。即使 x 非常小，也能正确得到 log（1+x）的结果。

（5）动态生成 C 语言代码。计算表达式更快速。

（6）广泛的单元测试和自我验证。能检测和诊断许多类型的错误。

由于 Theano 是为深度学习中处理大型神经网络算法所需的计算而专门设计的，因此其效率比其他深度学习框架要低，比较适合研究人员使用，不适合在线部署。

参考文献

[1] https://github.com/fchollet/keras/wiki/Keras-2.0-release-notes

[2] https://github.com/fchollet/keras/wiki/Keras

第 2 章

Keras 的安装与配置

第 1 章我们了解了 Keras 的技术背景、特点以及基本模型的构成，在使用 Keras 设计深度学习网络时，需要掌握 Keras 的基本安装与配置。

本章将介绍 Keras 的安装与配置，并详细介绍在 Windows 和 Linux 环境下的安装过程和配置方法。

2.1 Windows 环境下安装 Keras

2.1.1 硬件配置

1. 推荐配置

- 主板：X99 型号、Z270 型号。
- CPU：i7-6850K 或 i7-7500K 及以上型号。
- 内存：品牌内存，容量在 32GB 以上，根据主板组成 4 通道或 8 通道。

- SSD：品牌固态硬盘，容量在 256GB 以上。
- 显卡：NVIDIA GTX1080ti、NVIDIA GTX TITAN（Pascal）、NVIDIA GTX1080、NVIDIA GTX1070、NVIDIA GTX1060（顺序为优先性价比建议，并且建议同一显卡，可以根据主板插槽数量购买多块，例如 X99 型号主板最多可以采用×4 的显卡）。
- 电源：一般电源功耗为显卡功耗加 200W 即可。

2. 最低配置

- CPU：Intel 第四代 i5 和 i7 以上系列产品或同性能 AMD 公司产品。
- 显卡：计算能力高于 NVIDIA GTX750Ti 的 NVIDIA 显卡。
- 内存：容量在 8GB 以上。

3. 显卡建议

- 如果显卡是非 NVIDIA 公司的产品，或是 NVIDIA GTX 系列中型号的第一个数字低于或等于 4。或是 NVIDIA GT 系列，则不建议采用此类显卡进行加速计算，例如 NVIDIA GT910、NVIDIA GTX450 等。
- 如果显卡为笔记本上的 GTX 移动显卡（型号后面带有标识 M），那么请慎重使用显卡加速，因为移动版 GPU 很容易发生过热烧毁现象。
- 如果显卡显示的是诸如 HD5000、ATI5650 等类型的显卡，那么只能使用 CPU 计算。

2.1.2 Windows 版本

建议对于高性能的机器采用 Windows 10 作为基础环境，笔记本或低性能机器采用 Windows 7 即可。对于 Windows 10 的发行版本选择，笔者建议采用 Windows_10_Enterprise_2016_ltsb_x64 作为基础环境。

2.1.3 Microsoft Visual Studio 版本

使用 GPU 加速时，由于 CUDA 需要安装 Microsoft Visual Studio 作为编译器，因此需要安装 Microsoft Visual Studio 2010—2015。若没有 GPU 加速，则可跳过此部分安装。

CUDA 7.5 仅支持 2010、2012 和 2013 的 Visual Studio 版本，CUDA 8.0 仅支持 Visual Studio 2015 版本，本文采用 Visual Studio 2015 Update3。

2.1.4 Python 环境

Python 环境推荐使用科学计算集成 Python 发行版 Anaconda。Anaconda 是 Python 众多发行版中非常适用于科学计算的版本，里面已经集成了很多优秀的科学计算 Python 库。建议安装 Anaconda3 4.2.0 版本，因为目前新出的 Anaconda 与 Python 3.6 存在部分不兼容问题，所以建议安装历史版本 4.2.0。

友情提示：Windows 版本下的 TensorFlow 暂时不支持 Python 2.7。

2.1.5 CUDA

CUDA Toolkit 是 NVIDIA 公司面向 GPU 编程提供的基础工具包，也

是驱动显卡计算的核心技术工具。用户直接安装 CUDA 8.0 即可，下载地址为：https://developer.nvidia.com/cuda-downloads。在下载之后，按照步骤提示进行安装。不建议新手修改安装目录，环境不需要配置，安装程序会自动配置完毕。

2.1.6　加速库 CuDNN

从官网下载需要注册账号申请，约两三天能获得批准。通过网盘搜索一般也能找到最新版。Windows 对应的最新版目前是 cudnn-8.0-win-x64-v5.1-prod.zip。下载解压后是名为 cuda 的文件夹，里面有 bin、include 和 lib 3 个文件夹，需将这 3 个文件夹复制到安装 CUDA 的路径并覆盖对应的文件夹，默认文件夹为 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA。

2.1.7　Keras 框架的安装

在 CMD 命令行或者 Powershell 中输入：

```
    # GPU 版本
    >>>pip install --upgrade https://storage.googleapis.com/
tensorflow/windows/gpu/tensorflow_gpu-1.0.0-cp35-cp35m-win_am
d64.whl

    # CPU 版本
    >>> pip install --upgrade https://storage.googleapis.com/
tensorflow/windows/cpu/tensorflow-1.0.0-cp35-cp35m-win_amd64.
whl

    # Keras 安装
    >>> pip install keras -U -pre
```

之后可以验证 Keras 是否安装成功，在命令行中输入 Python 命令进入 Python，命令行环境如下：

```
>>> import keras
Using Tensorflow backend.
```

如果没有报错，那么表明 Keras 已经安装成功了。

2.2 Linux 环境下的安装

2.2.1 硬件配置

1. 推荐配置

- 主板：X99 型号、Z270 型号。
- CPU：i7-5830K 或 i7-6700K 及其以上型号。
- 内存：品牌内存，容量在 32GB 以上，根据主板组成 4 通道或 8 通道。
- SSD：品牌固态硬盘，容量在 256GB 以上。
- 显卡：NVIDIA GTX1080ti、NVIDIA GTX TITAN、NVIDIA GTX1080、NVIDIA GTX1070、NVIDIA GTX1060（顺序为优先建议，并且建议同一显卡，可以根据主板插槽数量购买多块，例如 X99 型号主板最多可以采用×4 的显卡）。
- 电源：一般电源功耗为显卡功耗加 200W 即可。

2. 最低配置

- CPU：Intel 第三代 i5 和 i7 以上系列产品或同性能 AMD 公司产品。
- 内存：总容量在 4GB 以上。

3. 显卡建议

- 如果您的显卡是非 NVIDIA 公司的产品，或是 NVIDIA GTX 系列中型号的第一个数字低于 6，或是 NVIDIA 的 GT 系列，则不建议采用此类显卡进行加速计算，例如 NVIDIA GT910、NVIDIA GTX450 等。
- 如果显卡为笔记本上的 GTX 移动显卡（型号后面带有标识 M），那么请慎重使用显卡加速，因为移动版 GPU 容易发生过热烧毁现象。
- 如果显卡显示的是诸如 HD5000、ATI5650 等类型的显卡，那么只能使用 CPU 计算。
- 如果显卡芯片为 Pascal 架构（NVIDIA GTX1080、NVIDIA GTX1070 等），需要在配置中再选择 CUDA 8.0。

2.2.2 Linux 版本

Linux 有很多发行版，强烈建议读者采用新版的 Ubuntu 16.04LTS。一方面，对于大多数新手来说，Ubuntu 具有很好的图形界面与丰富的开源社区；另一方面，Ubuntu 是 NVIDIA 官方的开发环境，同时也是绝大多

数深度学习框架默认的开发环境。不建议使用 Ubuntu 的其他版本，由于 GCC 编译器版本不同，会导致很多依赖无法有效安装。Ubuntu 16.04LTS 下载地址为：http://www.ubuntu.org.cn/download/desktop。

2.2.3 Ubuntu 环境的设置

安装开发包，打开终端输入：

```
    # 系统升级
    >>> sudo apt update
    >>> sudo apt upgrade
    # 安装 Python 基础开发包
    >>> sudo apt install -y python-dev python-pip python-nose gcc
g++ git gfortran vim
```

安装运算加速库，打开终端输入：

```
    >>> sudo apt install -y libopenblas-dev liblapack-dev
libatlas-base-dev
```

2.2.4 CUDA 开发环境

如果无 GPU 加速，则可跳过此安装步骤。CUDA 8.0 的下载地址：https://developer.nvidia.com/cuda-downloads。成功下载后，打开终端输入：

```
    >>> sudo dpkg -i cuda-repo-ubuntu1604-8-0-local_8.0.44-1_
amd64.deb
    >>> sudo apt update
    >>> sudo apt install cuda
```

自动配置成功就好，然后将 CUDA 路径添加至环境变量，即终端输入：

```
    >>> sudo gedit /etc/bash.bashrc
```

在 bash.bashrc 文件中添加：

```
export CUDA_HOME=/usr/local/cuda-8.0
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${LD_ LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

之后 source gedit /etc/bash.bashrc 即可。

同样，在终端输入：

```
>>> sudo gedit ~/.bashrc
```

在.bashrc 中添加如上相同的内容。

测试是否安装成功，在终端输入：

```
>>> nvcc -V
```

如果得到相应的 nvcc 编译器相应的信息，那么就表示 CUDA 配置成功了，需要重启系统。

2.2.5　加速库 cuDNN

从官网下载需要注册账号申请，需两三天批准。网盘搜索一般也能找到最新版。Linux 目前的版本是 cudnn-8.0-win-x64-v5.1-prod.zip。下载解压出来是名为 cuda 的文件夹，里面有 bin、include 和 lib 3 个文件夹，将这 3 个文件夹复制到安装 CUDA 的路径并覆盖对应的文件夹，在终端中输入：

```
>>>sudo cp include/cudnn.h /usr/local/cuda-8.0/include/
>>>sudo cp lib64/* /usr/local/cuda-8.0/lib64/
```

2.2.6 Keras 框架安装

相关开发包安装，在终端中输入：

```
>>>sudo pip install -U --pre pip setuptools wheel
>>>sudo pip install -U --pre numpy scipy matplotlib
scikit-learn scikit-image
>>>sudo pip install -U --pre tensorflow-gpu
# >>>sudo pip install -U --pre tensorflow
>>> sudo pip install -U --pre keras
```

安装完毕后，输入 Python，然后输入：

```
>>> import tensorflow
>>> import keras
```

若没有错误打印，表明安装成功。

第 3 章

Keras 快速上手

第 2 章我们详细介绍了在 Windows 和 Linux 环境下 Keras 的安装过程和配置方法，如何利用 Keras 构建深度学习网络是我们的目标。Keras 采用极简的设计理论、快速迭代的设计方法使得我们能够非常容易和快速地搭建神经网络模型，以达到在研究和实践中完成快速原型的需要。

如何快速使用 Keras 构建网络模型是本章的主要内容。本章通过实现一个 MNIST 手写数字识别案例，理解 Keras 深度学习框架的使用，帮助读者迅速掌握 Keras 的开发技巧，搭建自己的深度学习模型。

3.1 基本概念

（1）张量（Tensor）：张量是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型。

- 0 阶张量：即标量，也就是一个数值，是规模最小的张量。
- 1 阶张量：即向量，将若干数有序地排列起来就是向量。

- 2 阶张量：即矩阵，把一组向量有序的排列起来形成矩阵。
- 3 阶张量：也称为立方体，把多个矩阵叠起来就是 3 阶张量。例如 RGB 三种颜色通道的彩色图片就是 3 阶张量。
- 4 阶张量：继续把多个立方体叠起来，就是 4 阶张量，是一个纯数学上的概念。

张量的阶数有时候也称为维度，或者轴，轴这个词翻译自英文 axis。譬如一个矩阵[[1,2],[3,4]]，是一个 2 阶张量，有两个维度或轴，沿着第 0 个轴得到的是[1,2]和[3,4]两个向量，沿着第 1 个轴看到的是[1,3]和[2,4]两个向量。

下面的实例说明了维度概念，请读者参考并运行得到相应的结果。

```
import numpy as np

a = np.array([[1,2],[3,4]])
sum0 = np.sum(a, axis=0)
sum1 = np.sum(a, axis=1)

print sum0
print sum1
```

输出为：

```
[4 6]
[3 7]
```

（2）数据格式（Data Format）：是指对训练或测试数据的表达方法。比如一组彩色图片的数据表示，Theano 会把 100 张 RGB 三通道的 16×32（高为 16，宽为 32）彩色图表示为下面这种形式（100,3,16,32），其中第 0

个维度是样本维，代表样本的数目为 100；第 1 个维度是通道维，代表颜色通道数；后面两个就是高和宽了。Caffe 框架也采取了这种方式。我们把这种组织方法称为“channels first”，即通道维靠前。

而 TensorFlow 的表达形式是（100,16,32,3），即把通道维放在了最后。这种数据组织方式称为“channels last”，即通道维靠后。

Keras 默认的数据组织方式在~/.keras/keras.json 中定义，读者可查看此文件 image_data_format 项，也可以通过代码 K.image_data_format()函数返回得到数据组织方式。无论是哪种组织方式都是可行的，只要全局保持一致即可。

（3）函数式模型（Functional Model）：在 Keras 0.x 中，模型有两种。

- 第一种叫 Sequential，称为序贯模型，也就是单输入单输出，层与层之间只有相邻关系，跨层连接统统没有。这种模型编译速度快，操作上也比较简单。
- 第二种模型称为 Graph，即图模型，这个模型支持多输入多输出，层与层之间想怎么连就怎么连，但是编译速度慢。可以看到，Sequential 其实是 Graph 的一个特殊情况。

在 Keras 1 和 Keras 2 中，图模型被删除，增加了“functional model API”。此 API 更加强调了 Sequential 是特殊情况这一点，而一般的模型就称为 Model。由于“functional model API”在使用时利用的是“函数式编程”的风格，因此可称之为函数式模型。

（4）batch（批数据）：在解释这一名词之前，先了解一下深度学习的

优化算法，即梯度下降时，每次参数更新有两种方式。

- 第一种，遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为 Batch Gradient Descent，批梯度下降。

- 第二种，每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降（Stochastic Gradient Descent）[1]，如图 3-1 所示。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近抖动，命中不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

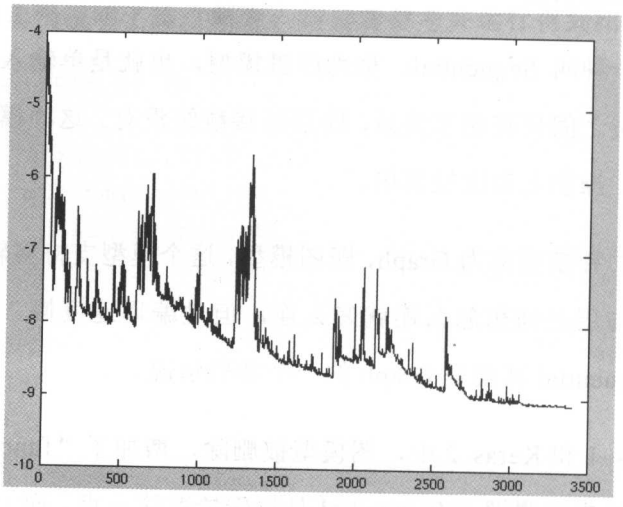


图 3-1　SGD 随机剃度下降法

为了克服两种方法的缺点，现在一般采用的是一种折中手段，Mini-batch Gradient Decent（小批梯度下降）。这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的

方向，下降起来就不容易跑偏，减少了随机性。另一方面，因为批的样本数与整个数据集相比小了很多，计算量也减少了很多。所以简单思考即可得出结论：batch 尺寸越大，训练时表现得越像 Batch Gradient Descent；batch 尺寸越小，训练时表现得越像 SGD，训练时抖动也越明显。

一般目前的梯度下降都是基于 Mini-batch 的，所以 Keras 的模块中经常会出现 batch_size，正是指 Mini-batch 的大小。

说明：Keras 中用的优化器 SGD 是 Stochastic Gradient Descent 的缩写，但不代表是一个样本就更新一回，SGD 仍然是基于 Mini-batch 的，并且 Mini-batch 逐渐成为训练时的“标配”。

（5）epochs：指的就是训练过程中整个数据集将被循环训练多少次。

3.2　初识 Sequential 模型

Keras 的核心数据结构是“模型”，模型是一种组织网络层的方式。Keras 中主要的模型是 Sequential 模型。Sequential 是一系列网络层按顺序构成的栈。Sequential 模型如下：

```
from keras.models import Sequential
model = Sequential()
```

将一些网络层通过.add()堆叠起来，就构成了一个模型。

```
from keras.layers.core import Dense, Dropout, Activation

model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
```

完成模型的搭建后，需要使用.compile()方法来编译模型。

```
    # 使用交叉熵作为 loss 函数
    model.compile(loss='categorical_crossentropy', optimizer=
sgd)
```

编译模型时，必须指明损失函数和优化器，如果需要的话，也可以自定义损失函数。Keras 的一个核心理念就是简明易用，同时，保证用户对 Keras 的绝对控制力度。用户可以根据需要定制自己的模型、网络层，甚至修改源代码。此例中，使用交叉熵作为 Loss 函数，优化器为 SGD 随机梯度下降法。

完成模型编译后，在训练数据上按 batch 进行一定次数的迭代来训练网络。

```
    model.fit(data, label, batch_size=100,
    nb_epoch=10,shuffle=True,verbose=1,validation_split=0.2)
```

随后，可以使用一行代码对我们的模型进行评估，看看模型的指标是否满足我们的要求。

```
    model.evaluate(data, label, batch_size=100, show_accuracy =
True, verbose=1)
```

至此，我们可以看到用 Keras 用来搭建深度学习模型是非常快捷的。

3.3 一个 Mnist 手写数字实例

3.3.1 Mnist 数据准备

Keras 中的 Mnist 数据集已经被划分成 60 000 个训练集，10 000 个测

试集的形式，按以下格式调用即可。

```
(X_train, y_train), (X_test, y_test) =.load_data()
```

将 X_train 和 X_test 由三维向量 60 000×28×28 转换成二维向量 60 000×784，进行 reshape 操作，如下：

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

接下来再将 X_train 和 X_test 的数据格式转换为 float32 存储，并进行归一化处理。

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

将类别向量（从 0 到 nb_classes 的整数向量）映射为二值类别矩阵，相当于将向量用 one-hot 形式重新编码。

```
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

3.3.2　建立模型

利用 Keras 建立 Sequential 模型。

```
from keras.models import Sequential
model = Sequential()
```

利用 Keras 建立第一个全连接层，共有 512 个神经元，激活函数为 ReLu，Dropout 比例为 0.2。

```
from keras.layers.core import Dense, Dropout, Activation
```

```
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
```

利用 Keras 建立第二个隐层，共有 512 个神经元，激活函数为 ReLu，Dropout 比例为 0.2。

```
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
```

输出层共有 10 个神经元，激活函数使用 SoftMax，得到分类结果如下。

```
model.add(Dense(10))
model.add(Activation('softmax'))
```

输出模型整体信息，参数总数量为 784×512+512 + 512×512+512 + 512×10+10 = 669 706。

```
model.summary()
```

3.3.3 训练模型

compile 接收 3 个参数，分别如下。

（1）优化器 optimizer：参数可指定为已预定义的优化器名，如 rmsprop、adagrad，或一个 Optimizer 类对象，如此处的 RMSprop()。

（2）损失函数 loss：参数为模型试图最小化的目标函数，可为预定义的损失函数，如 categorical_crossentropy、mse，也可以为一个损失函数。

（3）指标列表：对于分类问题，一般将该列表设置为 metrics=['accuracy']

```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])
```

训练模型时指定的参数如下。

- batch_size：指定梯度下降时，每个 batch 包含的样本数。
- nb_epoch：训练的轮数，即迭代次数。
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为 epoch 输出一行记录。
- validation_data：指定验证集。

fit 函数返回一个 History 的对象，其 History.history 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况。

```
history = model.fit(X_train, Y_train,
                    batch_size = batch_size,
                    nb_epoch = nb_epoch,
                    verbose = 1,
                    validation_data = (X_test, Y_test))
```

按 batch 计算在某些输入数据上的模型误差：

```
score = model.evaluate(X_test, Y_test, verbose=0)
```

训练部分输出如下：

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s - loss:
0.2468 - acc: 0.9245 - val_loss: 0.1062 - val_acc: 0.9662
Epoch 2/20
```

```
    60000/60000 [==============================] - 3s - loss:
0.1027 - acc: 0.9687 - val_loss: 0.0885 - val_acc: 0.9744
    Epoch 3/20
    60000/60000 [==============================] - 3s - loss:
0.0755 - acc: 0.9772 - val_loss: 0.0798 - val_acc: 0.9763
    Epoch 4/20
    60000/60000 [==============================] - 3s - loss:
0.0617 - acc: 0.9810 - val_loss: 0.1023 - val_acc: 0.9692
    Epoch 5/20
    60000/60000 [==============================] - 3s - loss:
0.0512 - acc: 0.9847 - val_loss: 0.0832 - val_acc: 0.9791
    Epoch 6/20
    60000/60000 [==============================] - 3s - loss:
0.0447 - acc: 0.9866 - val_loss: 0.0778 - val_acc: 0.9796
    Epoch 7/20
    60000/60000 [==============================] - 3s - loss:
0.0392 - acc: 0.9883 - val_loss: 0.0822 - val_acc: 0.9798
    Epoch 8/20
    60000/60000 [==============================] - 3s - loss:
0.0336 - acc: 0.9899 - val_loss: 0.0784 - val_acc: 0.9820
    Epoch 9/20
    60000/60000 [==============================] - 3s - loss:
0.0336 - acc: 0.9904 - val_loss: 0.0937 - val_acc: 0.9809
    Epoch 10/20
    60000/60000 [==============================] - 3s - loss:
0.0293 - acc: 0.9917 - val_loss: 0.0802 - val_acc: 0.9829
    Epoch 11/20
    60000/60000 [==============================] - 3s - loss:
0.0260 - acc: 0.9924 - val_loss: 0.0966 - val_acc: 0.9821
    Epoch 12/20
    60000/60000 [==============================] - 3s - loss:
0.0240 - acc: 0.9932 - val_loss: 0.0984 - val_acc: 0.9836
    Epoch 13/20
    60000/60000 [==============================] - 3s - loss:
0.0230 - acc: 0.9939 - val_loss: 0.1032 - val_acc: 0.9822
    Epoch 14/20
    60000/60000 [==============================] - 3s - loss:
```

```
0.0236 - acc: 0.9933 - val_loss: 0.1002 - val_acc: 0.9843
    Epoch 15/20
    60000/60000 [==============================] - 3s - loss:
0.0184 - acc: 0.9945 - val_loss: 0.1111 - val_acc: 0.9811
    Epoch 16/20
    60000/60000 [==============================] - 3s - loss:
0.0201 - acc: 0.9944 - val_loss: 0.0982 - val_acc: 0.9837
    Epoch 17/20
    60000/60000 [==============================] - 3s - loss:
0.0186 - acc: 0.9949 - val_loss: 0.1012 - val_acc: 0.9841
    Epoch 18/20
    60000/60000 [==============================] - 3s - loss:
0.0179 - acc: 0.9951 - val_loss: 0.1132 - val_acc: 0.9824
    Epoch 19/20
    60000/60000 [==============================] - 3s - loss:
0.0189 - acc: 0.9950 - val_loss: 0.1081 - val_acc: 0.9842
    Epoch 20/20
    60000/60000 [==============================] - 3s - loss:
0.0168 - acc: 0.9956 - val_loss: 0.1109 - val_acc: 0.9837
```

最后输出训练好的模型的测试集上的表现如下：

```
    print('Test score:', score[0])
    print('Test accuracy:', score[1])
```

至此，我们掌握了利用 Keras 实现一个从数据准备、模型建立、模型训练和模型评估的完整方法，对 Keras 的特点有了初步的认识。

参考文献

[1]　http://sebastianruder.com/optimizing-gradient-descent/

第 4 章

Keras 模型的定义

第 3 章我们通过实现一个 MNIST 手写数字识别的案例，理解了 Keras 深度学习框架的使用，并初步掌握了 Keras 开发技巧。Keras 深度学习框架有哪些基本的模型？具体有哪些接口定义和特点？

本章详细介绍 Keras 的两种类型模型：Sequential 模型和函数式模型，以及它们的参数定义和接口，并说明了使用中应该注意的地方。

4.1 Keras 模型

Keras 有两种类型的模型：序贯模型（Sequential）和函数式模型（Model）。函数式模型应用更为广泛，Sequential 模型是函数式模型的一种特殊情况。

Keras 模型中常用的方法有以下几种。

- model.summary()：表示打印出模型的概况。
- model.get_config()：表示返回包含模型配置信息的 Python 字典。

模型可以从config信息中重构回去，其中config是一个类似JSON格式的字典对象。

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential:
model = Sequential.from_config(config)
```

- model.get_layer()：表示依据层名或下标获得层对象。
- model.get_weights()：表示返回模型权重张量的列表，类型为NumPy array。
- model.set_weights()：表示从 NumPy array 里将权重载入给模型，要求数组具有与 model.get_weights()相同的形状。
- model.to_json：表示返回代表模型的 JSON 字符串，仅包含网络结构，不包含权值。用户可以从 JSON 字符串中重构原模型。

```
from models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

model.to_yaml 与 model.to_json 类似，同样可以从产生的 YAML 字符串中重构模型。

```
from models import model_from_yaml

yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

- model.save_weights(filepath)：表示将模型权重保存到指定路径，文件类型是 HDF5（后缀是.h5）。

- model.load_weights(filepath, by_name=False)：表示从 HDF5 文件中加载权重到当前模型中，默认情况下模型的结构将保持不变。如果想将权重载入不同的模型（有些层相同）中，则设置 by_name=True，只有名字匹配的层才会载入权重。

4.2 Sequential 模型

Sequential 模型是多个网络层的线性堆叠，可以通过向 Sequential 模型传递一个 Layer List 构造此模型。

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
Dense(32, units=784),
Activation('relu'),
Dense(10),
Activation('softmax'),
])
```

也可以通过 add()方法依次将 Layer 添加到模型中。

```
model = Sequential()
model.add(Dense(32, units=784))
model.add(Activation('relu'))
```

4.2.1 Sequential 模型接口

（1）add：表示向模型中添加一个层。

```
add(self, layer)
```

layer 为 Layer 对象；

（2）pop：表示弹出模型的最后一层，无返回值。

```
pop(self)
```

（3）compile：表示编译用来配置模型的学习过程。

```
compile(self, optimizer, loss, metrics=None, sample_weight_
mode=None)
```

其参数有以下几个。

- optimizer：表示字符串（预定义优化器名）或优化器对象，参考优化器。
- loss：表示字符串（预定义损失函数名）或目标函数，参考损失函数。
- metrics：表示列表，包含评估模型在训练和测试时的网络性能的指标，典型用法是 metrics=['accuracy']。
- sample_weight_mode：如果需要按时间步为样本赋权（2D 权矩阵），则将该值设为“temporal”。默认为“None”，代表按样本赋权（1D 权）。用户可参考 fit 函数的解释中此参数的详细说明。
- kwargs：使用 TensorFlow 作为后端可忽略该参数，若使用 Theano 作为后端，kwargs 的值将会传递给 K.function。

```
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
```

```
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

模型在使用前必须编译，否则在调用 fit 或 evaluate 时会抛出异常。

（4）fit 函数：具体形式如下。

```
    fit(self, x, y, batch_size=32, epochs=10, verbose=1,
callbacks=None, validation_split=0.0, validation_data=None,
shuffle=True, class_weight=None, sample_weight=None, initial_
epoch=0)
```

fit 函数将模型训练 nb_epoch 轮，其参数有以下几种。

- x：输入数据。如果模型只有一个输入，那么 x 的类型是 NumPy array；如果模型有多个输入，那么 x 的类型应当为 List。List 的元素是对应于各个输入的 NumPy array。
- y：标签，NumPy array。
- batch_size：整数，指定进行梯度下降时每个 batch 包含的样本数。训练时一个 batch 的样本会被计算一次梯度下降，使目标函数优化一步。
- epochs：整数，训练的轮数，每个 epoch 会把训练集循环一遍。
- verbose：日志显示，设为 0，表示不在标准输出流输出日志信息；设为 1，表示输出进度条记录；设为 2，表示每个 epoch 输出一行记录。
- callbacks：表示 List 类，其中的元素是 keras.callbacks.Callback 的对象。这个 list 中的回调函数将会在训练过程中的适当时候被

调用，可参考回调函数。

- validation_split：表示 0~1 的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个 epoch 结束后测试的模型的指标，如损失函数、精确度等。注意，validation_split 的划分在 shuffle 之前，因此如果数据本身是有序的，需要先打乱再指定 validation_split，否则可能会出现验证集样本不均匀。

- validation_data：形式为（x，y）的 tuple，是指定的验证集。此参数将覆盖 validation_spilt。

- shuffle：布尔值或字符串，一般为布尔值，表示是否在训练过程中随机打乱输入样本的顺序。若为字符串“batch”，则是用来处理 HDF5 数据的特殊情况，它将在 batch 内部将数据打乱。

- class_weight：字典，将不同的类别映射为不同的权值。该参数用来在训练过程中调整损失函数（只能用于训练）。

- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练）。用户可以传递一个 1D 的与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 sample_weight_mode='temporal'。

- initial_epoch: 从该参数指定的 epoch 开始训练，在继续之前的训

练时有用。

fit 函数返回一个 History 的对象，其 History.history 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况。

（5）evaluate 函数：表示按 batch 计算在某些输入数据上模型的误差。

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight
=None)
```

evaluate 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：标签，NumPy array。
- batch_size：整数，含义同 fit 函数的同名参数。
- verbose：含义同 fit 函数的同名参数，但只能取 0 或 1。
- sample_weight：NumPy array，含义同 fit 函数的同名参数。

evaluate 函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的 list（如果模型还有其他的评价指标）。model.metrics_names 将给出 list 中各个值的含义。

（6）predict 函数：按 batch 获得输入数据对应的输出。

```
predict(self, x, batch_size=32, verbose=0)
```

predict 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- batch_size：整数，含义同 fit 函数的同名参数。
- verbose：含义同 fit 函数的同名参数，但只能取 0 或 1。

predict 函数的返回值是预测值的 NumPy array。

（7）predict_proba 函数：按 batch 产生输入数据的类别预测结果。

```
predict_proba(self, x, batch_size=32, verbose=1)
```

predict_proba 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- batch_size：整数，含义同 fit 函数的同名参数。
- verbose：含义同 fit 函数的同名参数，但只能取 0 或 1。

predict_proba 函数的返回值是类别概率的 NumPy array。

（8）train_on_batch 函数：在一个 batch 的数据上进行一次参数更新。

```
train_on_batch(self, x, y, class_weight=None, sample_weight
=None)
```

train_on_batch 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。

- y：标签，NumPy array。
- class_weight：含义同 fit 函数的同名参数。
- sample_weight：含义同 fit 函数的同名参数。

train_on_batch 函数返回训练误差的标量值或标量值的 list，与 evaluate 的情形相同。

（9）test_on_batch 函数：在一个 batch 的样本上对模型进行评估。

```
test_on_batch(self, x, y, sample_weight=None)
```

test_on_batch 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：标签，Numpy array。
- sample_weight：含义同 fit 函数的同名参数。

test_on_batch 函数的返回与 evaluate 的情形相同。

（10）predict_on_batch 函数：在一个 batch 的样本上对模型进行测试。

```
predict_on_batch(self, x)
```

predict_on_batch 函数的参数如下。

x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。

predict_on_batch 函数返回模型在一个 batch 上的预测结果。

（11）fit_generator 函数：利用 Python 的生成器，逐个生成数据的 batch

并进行训练。生成器与模型将并行执行，以提高效率。

```
    fit_generator(self, generator, steps_per_epoch, epochs=1,
verbose=1, callbacks=None, validation_data=None, validation_
steps=None,   class_weight=None,   max_q_size=10,   workers=1,
pickle_safe=False, initial_epoch=0)
```

例如，该函数允许在 CPU 上进行实时的数据提升，同时在 GPU 上进行模型训练，其参数有以下几种。

- generator：生成器函数，生成器的输出有两种形式。
  - ✧ 一个形如（inputs，targets）的 tuple；
  - ✧ 一个形如（inputs, targets,sample_weight）的 tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个 epoch 以经过模型的样本数达到 samples_per_epoch 时，记一个 epoch 结束。
- steps_per_epoch：整数，表示当生成器返回 steps_per_epoch 次数据时计一个 epoch 结束，执行下一个 epoch。
- epochs：整数，表示数据迭代的轮数。
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录。
- validation_data：具有以下 3 种形式之一。
  - ✧ 生成验证集的生成器；
  - ✧ 一个形如（inputs,targets）的 tuple；

✧ 一个形如（inputs,targets，sample_weights）的 tuple。

- validation_steps：当 validation_data 为生成器时，本参数用于指定验证集的生成器返回次数。
- class_weight：表示规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡的问题。
- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋予不同的权。这种情况下请确定在编译模型时添加了 sample_weight_mode='temporal'。
- workers：表示最大进程数。
- max_q_size：表示生成器队列的最大容量。
- pickle_safe：若为 True，则使用基于进程的线程。由于该实现依赖多进程，不能传递 non picklable（无法被 pickle 序列化）的参数到生成器中，因为无法轻易将它们传入子进程中。
- initial_epoch：从该参数指定的 epoch 开始训练，在继续之前的训练时有用。本函数返回一个 History 对象

（12）evaluate_generator 函数：使用一个生成器作为数据源评估模型，生成器应返回与 test_on_batch 的输入数据相同类型的数据。

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight
```

```
= None)
```

evaluate_generator 函数的参数与 fit_generator 同名参数含义相同，steps 是生成器要返回数据的轮数，其参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：表示标签，NumPy array。
- batch_size：整数，含义同 fit_generator 的同名参数。
- verbose：含义同 fit_generator 的同名参数，但只能取 0 或 1。
- sample_weight：含义同 fit_generator 的同名参数。

（13）predcit_generator 函数：使用一个生成器作为数据源预测模型，生成器应返回与 test_on_batch 的输入数据相同类型的数据。

```
predict_generator(self, generator, steps, max_q_size=10,
workers=1, pickle_safe=False, verbose=0)
```

predcit_generator 函数的参数与 fit_generator 同名参数含义相同，steps 是生成器要返回数据的轮数，其参数有以下几种。

- generator：生成器函数，含义同 fit_generator 的同名参数。
- max_q_size：表示生成器队列的最大容量，含义同 fit_generator 的同名参数。
- workers：表示最大进程数，含义同 fit_generator 的同名参数。
- pickle_safe：若为 True，则使用基于进程的线程，含义同

fit_generator 的同名参数。

- verbose：含义同 fit_generator 的同名参数，但只能取 0 或 1。

4.2.2 Sequential 模型的数据输入

Keras 模型需要定义输入数据的 Shape，Sequential 模型的第一层需要接收一个关于输入数据 shape 的参数，后面的各个层则可以自动地推导出中间数据的 shape，因此不需要为每个层都指定这个参数。下面介绍几种方法来为第一层指定输入数据的 shape。

- 传递一个 input_shape 的关键字参数给第一层，input_shape 是一个 tuple 类型的数据，其中也可以输入 None。如果输入 None，则表示此位置可能是任何正整数。数据的 batch 大小不应包含在其中。

- 某些 2D 层，如 Dense 层，支持通过指定其输入维度 input_dim 来隐含的指定输入数据 shape。一些 3D 的时域层支持通过参数 input_dim 和 input_length 来指定输入 shape。

- 如果需要为输入指定一个固定大小的 batch_size（常用于 stateful RNN 网络），可以传递 batch_size 参数到一个层中，例如需要指定输入张量的 batch 大小是 32，数据 shape 是（6，8），则传递 batch_size=32 和 input_shape=(6,8)。

下面的两个指定输入数据 shape 的方法是严格等价的：

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, input_shape=(784))
```

4.2.3　模型编译

在训练模型之前，通过 compile 来对学习过程进行配置。compile 接收 3 个参数，分别如下。

- optimizer（优化器）：该参数可指定为已预定义的优化器名，如 rmsprop、adagrad 或一个 Optimizer 类的对象。
- loss（损失函数）：该参数是模型试图最小化的目标函数，它可设为预定义的损失函数名，如 categorical_crossentropy、mse，也可以设为一个损失函数。
- metrics（性能评估指标列表）：对分类问题，我们一般将该列表设置为 metrics=['accuracy']。该指标既可以是一个预定义指标的名字，也可以是一个用户定制的函数。该指标函数应该返回单个张量，或一个完成 metric_name → metric_value 映射的字典。

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
```

```
    model.compile(optimizer='rmsprop',
                  loss='mse')

    # For custom metrics
    import keras.backend as K

    def mean_pred(y_true, y_pred):
        return K.mean(y_pred)

    model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy', mean_pred])
```

4.2.4 模型训练

Keras 以 NumPy 数组作为输入数据和标签的数据类型。训练模型一般使用 fit 函数，例如：

```
    # For a single-input model with 2 classes (binary
classification):

    model = Sequential()
    model.add(Dense(32, activation='relu', input_dim=100))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    # Generate dummy data
    import numpy as np
    data = np.random.random((1000, 100))
    labels = np.random.randint(2, size=(1000, 1))

    # Train the model, iterating on the data in batches of 32
samples
    model.fit(data, labels, epochs=10, batch_size=32)
```

```
    # For a single-input model with 10 classes (categorical
classification):

    model = Sequential()
    model.add(Dense(32, activation='relu', input_dim=100))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Generate dummy data
    import numpy as np
    data = np.random.random((1000, 100))
    labels = np.random.randint(10, size=(1000, 1))

    # Convert labels to categorical one-hot encoding
    one_hot_labels = keras.utils.to_categorical(labels, num_
classes = 10)

    # Train the model, iterating on the data in batches of 32
samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

4.3　函数式模型

Keras 函数式模型接口是用户定义多输出模型、非循环有向模型或具有共享层模型等复杂模型的途径。当模型需要多于一个的输出，应该选择函数式模型。函数式模型是最广泛的一类模型，Sequential 模型只是它的一种特殊情况。

4.3.1 全连接网络

从一个例子开始：

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

- 层对象接受张量为参数，返回一个张量。
- 输入是张量，输出也是张量的一个网络就是一个模型，可通过 Model 定义。
- 这样的模型可以被像 Keras 的 Sequential 一样被训练。

利用函数式模型的接口，可以很容易地重用已经训练好的模型：可以把模型当作一个层一样，通过提供一个张量来调用它。当调用一个模型时，不仅仅重用了它的结构，也重用了它的权重。

```
x = Input(shape=(784,))
```

```
    # This works, and returns the 10-way softmax we defined above.
y = model(x)
```

这种方式可以快速地创建能处理序列信号的模型，将一个图像分类的模型变为一个对视频分类的模型，只需要一行代码。

```
    from keras.layers import TimeDistributed

    # Input tensor for sequences of 20 timesteps,
    # each containing a 784-dimensional vector
    input_sequences = Input(shape=(20, 784))

    # This applies our previous model to every timestep in the
input sequences.
    # the output of the previous model was a 10-way softmax,
    # so the output of the layer below will be a sequence of 20
vectors of size 10.
    processed_sequences = TimeDistributed(model)(input_sequences)
```

4.3.2　函数模型接口

Keras 的函数式模型为 Model，即广义的拥有输入和输出的模型。使用 Model 来初始化一个函数式模型：

```
    from keras.models import Model
    from keras.layers import Input, Dense

    a = Input(shape=(32,))
    b = Dense(32)(a)
    model = Model(inputs=a, outputs=b)
```

在这里，模型以 a 为输入，以 b 为输出，同样可以构造拥有多输入和多输出的模型。

```
    model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])
```

Model 属性如下。

- model.layers：表示组成模型图的各个层。
- model.inputs：表示模型的输入张量列表。
- model.outputs：表示模型的输出张量列表。

Model 模型方法有以下几种。

（1）compile 函数：编译模型以供训练。

```
compile(self, optimizer, loss, metrics=None, loss_weights =
None, sample_weight_mode=None)
```

compile 函数的参数有以下几种。

- optimizer：优化器，为预定义优化器名或优化器对象，参考优化器。
- loss：损失函数，为预定义损失函数名或一个目标函数，参考损失函数。
- metrics：列表，包含评估模型在训练和测试时的性能的指标，典型用法是 metrics=['accuracy']。如果要在多输出模型中为不同的输出指定不同的指标，可像该参数传递一个字典，例如 metrics = {'ouput_a': 'accuracy'}。
- sample_weight_mode：如果需要按时间步为样本赋权（2D 权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D 权）。如果模型有多个输出，可以向该参数传入指定

sample_weight_mode 的字典或列表。

- kwargs：使用 TensorFlow 作为后端请忽略该参数。若使用 Theano 作为后端，kwargs 的值将会传递给 K.function。

当参数传入非法值时会抛出异常。

提示：如果只载入模型并利用其 predict，可以不用进行 compile。在 Keras 中，compile 主要完成损失函数和优化器的一些配置，是为训练服务的。predict 会在内部进行符号函数的编译工作（通过调用_make_predict_function 生成函数）。

（2）fit 函数：用于训练模型。

```
fit(self, x=None, y=None, batch_size=32, epochs=1, verbose=1,
callbacks=None,  validation_split=0.0,  validation_data=None,
shuffle=True, class_weight=None, sample_weight=None, initial_
epoch=0)
```

fit 函数的参数有以下几种。

- x：表示输入数据。如果模型只有一个输入，那么 x 的类型是 NumPy array；如果模型有多个输入，那么 x 的类型应当为 List，List 的元素是对应于各个输入的 NumPy array。如果模型的每个输入都有名字，则可以传入一个字典，将输入名与其输入数据对应起来。

- y：标签，NumPy array。如果模型有多个输出，可以传入一个 NumPy array 的 List。如果模型的输出拥有名字，则可以传入一个字典，将输出名与其标签对应起来。

- batch_size：整数，指定进行梯度下降时每个 batch 包含的样本数。训练时一个 batch 的样本会被计算一次梯度下降，使目标函数优化一步。
- nb_epoch：整数，训练的轮数，训练数据将会被遍历 nb_epoch 次。
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录。
- callbacks：List 类型，其中的元素是 keras.callbacks.Callback 的对象。这个 list 中的回调函数将会在训练过程中的适当时候被调用。
- validation_split：表示 0~1 的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个 epoch 结束后测试的模型的指标，如损失函数、精确度等。注意，validation_split 的划分在 shuffle 之后，因此如果数据本身是有序的，需要先手工打乱再指定 validation_split，否则可能会出现验证集样本不均匀。
- validation_data：形式为（x，y）或（x，y，sample_weights）的 tuple，是指定的验证集。此参数将覆盖 validation_spilt。
- shuffle：布尔值，表示是否在训练过程中每个 epoch 前随机打乱输入样本的顺序。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）。该参数在处

理非平衡的训练数据（某些类的训练样本数很少）时，可以使得损失函数对样本数不足的数据更加关注。

- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 sample_weight_mode='temporal'。
- initial_epoch：从该参数指定的 epoch 开始训练，在继续之前的训练时有用；输入数据与规定数据不匹配时会抛出错误显示。

fit 函数返回一个 History 的对象，其 History.history 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况。

（3）evaluate 函数：按 batch 计算在某些输入数据上模型的误差。

```
    evaluate(self, x, y, batch_size=32, verbose=1, sample_weight
=None)
```

evaluate 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：标签，NumPy array。
- batch_size：整数，含义同 fit 函数的同名参数。

- verbose：含义同 fit 函数的同名参数，但只能取 0 或 1。
- sample_weight：NumPy array，含义同 fit 函数的同名参数。

evaluate 函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的 list（如果模型还有其他的评价指标）。model.metrics_names 将给出 list 中各个值的含义。

（4）predict 函数：按 batch 获得输入数据对应的输出。

```
predict(self, x, batch_size=32, verbose=0)
```

predict 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- batch_size：整数，含义同 fit 函数的同名参数。
- verbose：含义同 fit 函数的同名参数，但只能取 0 或 1。

predict 函数的返回值是预测值的 NumPy array。

（5）train_on_batch 函数：在一个 batch 的数据上进行一次参数更新。

```
train_on_batch(self, x, y, class_weight=None, sample_weight
= None)
```

train_on_batch 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：标签，NumPy array。

- class_weight：字典，将不同的类别映射为不同的权值，含义同 fit 函数的同名参数。
- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练），含义同 fit 函数的同名参数。

train_on_batch 函数返回训练误差的标量值或标量值的 List，与 evaluate 的情形相同。

（6）test_on_batch 函数：在一个 batch 的样本上对模型进行评估。

```
test_on_batch(self, x, y, sample_weight=None)
```

test_on_batch 函数的参数有以下几种。

- x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。
- y：标签，NumPy array。
- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练），含义同 fit 函数的同名参数。

test_on_batch 函数的返回与 evaluate 的情形相同。

（7）predict_on_batch 函数：在一个 batch 的样本上对模型进行测试。

```
predict_on_batch(self, x)
```

predict_on_batch 函数的参数如下。

x：输入数据，与 fit 函数一样，是 NumPy array 或 NumPy array 的 List。

predict_on_batch 函数返回模型在一个 batch 上的预测结果。

（8）fit_generator 函数：利用 Python 的生成器，逐个生成数据的 batch 并进行训练。生成器与模型将并行执行，以提高效率。

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbose=1, callbacks=None, validation_data=None, validation_steps=None, class_weight=None, max_q_size=10, workers=1, pickle_safe=False, initial_epoch=0)
```

例如，该函数允许在 CPU 上进行实时的数据提升，同时在 GPU 上进行模型训练，其参数有以下几种。

- generator：生成器函数，生成器的输出有以下两种情况。
  - ✧ 一个形如（inputs，targets）的 tuple；
  - ✧ 一个形如（inputs, targets,sample_weight）的 tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个 epoch 以经过模型的样本数达到 samples_per_epoch 时，记一个 epoch 结束。
- steps_per_epoch：整数，当生成器返回 steps_per_epoch 次数据时计一个 epoch 结束，执行下一个 epoch。
- epochs：整数，数据迭代的轮数。
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录。
- validation_data：具有以下 3 种形式之一。

- ✧ 生成验证集的生成器；
- ✧ 一个形如（inputs,targets）的 tuple；
- ✧ 一个形如（inputs,targets，sample_weights）的 tuple。

- validation_steps：当 validation_data 为生成器时，用于指定验证集的生成器返回次数。
- class_weight：规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡的问题。
- sample_weight：权值的 NumPy array，用于在训练时调整损失函数（仅用于训练）。它可以传递一个 1D 与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 sample_weight_mode='temporal'。
- workers：表示最大进程数。
- max_q_size：生成器队列的最大容量；
- pickle_safe: 若为 True，则使用基于进程的线程。由于该实现依赖多进程，不能传递 non picklable（无法被 pickle 序列化）的参数到生成器中，因为无法轻易将它们传入到子进程中。
- initial_epoch: 从该参数指定的 epoch 开始训练，在继续之前的训练时有用。

函数返回一个 History 对象。

（9）evaluate_generator 函数：使用一个生成器作为数据源，来评估模型，生成器应返回与 test_on_batch 的输入数据相同类型的数据。

```
    evaluate_generator(self, generator, steps, max_q_size=10,
workers=1, pickle_safe=False)
```

evaluate_generator 函数的参数有以下几种。

- generator：生成输入 batch 数据的生成器。
- val_samples：生成器应该返回的总样本数。
- steps：生成器要返回数据的轮数。
- max_q_size：生成器队列的最大容量。
- nb_worker：使用基于进程的多线程处理时的进程数。
- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易地传递到子进程中。

（10）predict_generator 函数：从一个生成器上获取数据并进行预测，生成器应返回与 predict_on_batch 输入类似的数据。

```
    fit_generator(self, generator, steps_per_epoch, epochs=1,
verbose=1, callbacks=None, validation_data=None, validation_
steps=None,   class_weight=None,   max_q_size=10,   workers=1,
pickle_safe=False, initial_epoch=0)
```

predict_generator 函数的参数有以下几种。

- generator：生成输入 batch 数据的生成器。
- val_samples：生成器应该返回的总样本数。
- max_q_size：生成器队列的最大容量。
- nb_worker：使用基于进程的多线程处理时的进程数。
- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易地传递到子进程中。

4.3.3　多输入和多输出模型

使用函数式模型的一个典型场景是搭建多输入、多输出的模型。有这样一个案例：希望预测 Twitter 上一条新闻会被转发和点赞的次数。模型的主要输入是新闻本身，也就是一个词语的序列。还可以拥有额外的输入，如新闻发布的日期等。这个模型的损失函数将由两部分组成，辅助的损失函数评估仅仅基于新闻本身的情况做出预测，而主损失函数评估基于新闻和额外信息进行预测，即使来自主损失函数的梯度发生弥散，来自辅助损失函数的信息也能够训练 Embeddding 和 LSTM 层。在模型中使用主要的损失函数是对于深度网络的一个良好的正则方法。总而言之，该模型框图如图 4-1 所示。

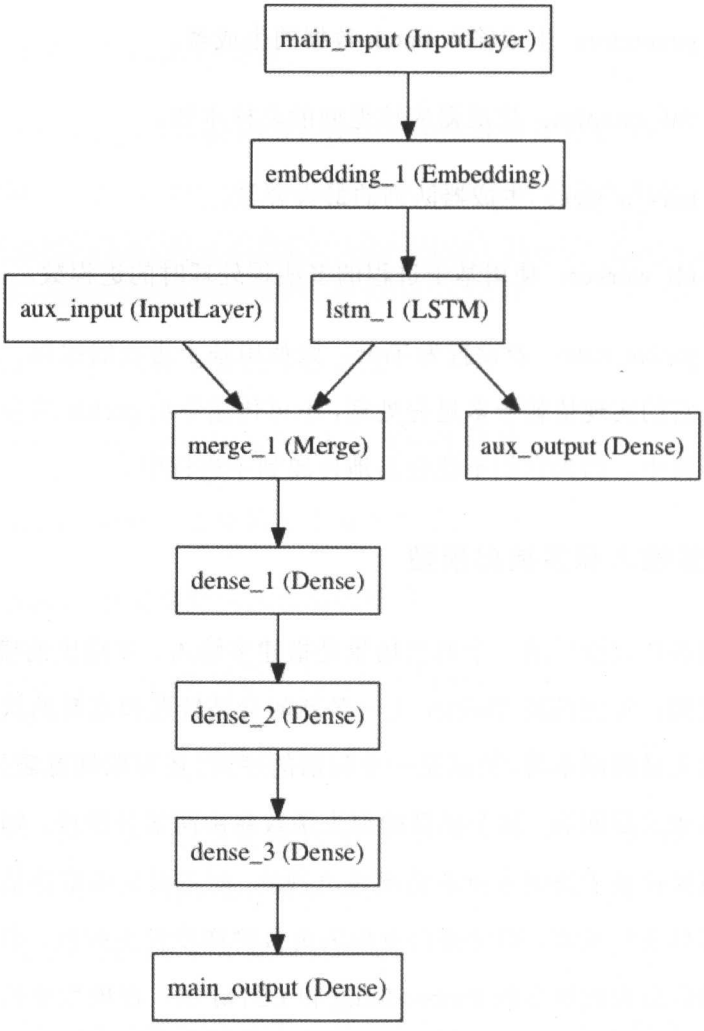


图 4-1　多输入和多输出模型框图

我们可以用函数式模型来实现这个框图。主要的输入是接收新闻本身，即一个整数的序列（每个整数编码一个单词）。这些整数位于 1 到 10 000 之间（即我们的字典有 10 000 个词）。这个序列有 100 个单词。

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model
```

```
    # Headline input: meant to receive sequences of 100 integers,
between 1 and 10000.
    # Note that we can name any layer by passing it a "name"
argument.
    main_input = Input(shape=(100,), dtype='int32', name='main_
input')

    # This embedding layer will encode the input sequence
    # into a sequence of dense 512-dimensional vectors.
    x = Embedding(output_dim=512, input_dim=10000, input_
length=100)(main_input)

    # A LSTM will transform the vector sequence into a single
vector,
    # containing information about the entire sequence
    lstm_out = LSTM(32)(x)
```

然后，插入一个额外的损失函数，使得即使在主损失很高的情况下，LSTM 和 Embedding 层也可以平滑地训练。

```
    auxiliary_output = Dense(1, activation='sigmoid', name=
'aux_output')(lstm_out)
```

接下来，将 LSTM 与额外的输入数据串联起来组成输入，送入模型中。

```
    auxiliary_input = Input(shape=(5,), name='aux_input')
    x = keras.layers.concatenate([lstm_out, auxiliary_input])

    # We stack a deep densely-connected network on top
    x = Dense(64, activation='relu')(x)
    x = Dense(64, activation='relu')(x)
    x = Dense(64, activation='relu')(x)

    # And finally we add the main logistic regression layer
    main_output = Dense(1, activation='sigmoid', name = 'main_
output')(x)
```

最后，我们定义两个输入，两个输出的模型。

```
    model = Model(inputs=[main_input, auxiliary_input], outputs
=[main_output, auxiliary_output])
```

模型定义完毕，下一步就要编译模型。对额外的损失赋 0.2 的权重。用户可以通过关键字参数 loss_weights 或 loss 来为不同的输出设置不同的损失函数或权值。这两个参数均可为 Python 的列表或字典。这里给 loss 传递单个损失函数，这个损失函数会被应用于所有的输出上。

```
    model.compile(optimizer='rmsprop',
loss='binary_crossentropy',
                  loss_weights=[1., 0.2])
```

编译完成后，通过传递训练数据和目标值训练该模型。

```
    model.fit([headline_data, additional_data], [labels, labels],
              epochs=50, batch_size=32)
```

因为输入和输出是被命名过的（在定义时传递了“name”参数），可以用下面的方式编译和训练模型。

```
    model.compile(optimizer='rmsprop',
                  loss={'main_output':    'binary_crossentropy',
'aux_output': 'binary_crossentropy'},
                  loss_weights={'main_output': 1., 'aux_output':
0.2})

    # And trained it via:
    model.fit({'main_input':    headline_data,    'aux_input':
additional_data},
              {'main_output': labels, 'aux_output': labels},
              epochs=50, batch_size=32)
```

4.3.4　共享层模型

另一个使用函数式模型的场合是使用共享层的情况。考虑微博数据，希望建立模型来判别两条微博是否是来自同一个用户，这个需求同样可以用来判断一个用户的两条微博的相似性。

一种实现方式是，建立一个模型，它分别将两条微博的数据映射到两个特征向量上，然后将特征向量串联并加一个 logistic 回归层，输出它们来自同一个用户的概率。这种模型的训练数据是一对对的微博。

因为这个问题是对称的，所以处理第一条微博的模型当然也能重用于处理第二条微博，所以使用一个共享的 LSTM 层来进行映射。

首先，将微博的数据转为（140,256）的矩阵，即每条微博有 140 个字符，每个单词的特征由一个 256 维的词向量表示，向量的每个元素为 1，表示某个字符出现；为 0，则表示不出现，这是一个 one-hot 编码。

之所以是（140,256）是因为一条微博最多有 140 个字符，而扩展的 ASCII 码表编码了常见的 256 个字符。

```
import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它。

```
# This layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)
```

```
    # When we reuse the same layer instance
    # multiple times, the weights of the layer
    # are also being reused
    # (it is effectively *the same* layer)
    encoded_a = shared_lstm(tweet_a)
    encoded_b = shared_lstm(tweet_b)

    # We can then concatenate the two vectors:
    merged_vector    =    keras.layers.concatenate([encoded_a,
encoded_b], axis=-1)

    # And add a logistic regression on top
    predictions = Dense(1, activation='sigmoid')(merged_vector)

    # We define a trainable model linking the
    # tweet inputs to the predictions
    model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

    model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    model.fit([data_a, data_b], labels, epochs=10)
```

我们先关注一下共享层的实现，包括其输出数据以及输出数据的 shape。

当在某个输入上调用层时，就创建了一个新的张量（即该层的输出），同时在为这个层增加一个“（计算）节点”。这个节点将输入张量映射为输出张量。当多次调用该层时，这个层就有了多个节点，其下标分别为 0，1，2，……

在 Keras 中，可以通过 layer.get_output()方法来获得层的输出张量，或者通过 layer.output_shape 获得其输出张量的 shape。

如果层只与一个输入相连，那没有任何问题，对应的 output 将会返回该层唯一的输出。

```
a = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a
```

但当层与多个输入相连时，会出现问题，比如下面的一段代码：

```
a = Input(shape=(140, 256))
b = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output
```

出现错误信息如下：

```
>> AssertionError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.
```

这时，可通过下面这种调用方式即可解决。

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```

对于 input_shape 和 output_shape 也是一样，如果一个层只有一个节点，或所有的节点都有相同的输入或输出 shape，那么 input_shape 和 output_shape 都是没有歧义的，并也只返回一个值。但是，例如把一个相

同的 Conv2D 应用于一个大小为（3,32,32）的数据，然后又将其应用于一个（3,64,64）的数据，那么此时该层就具有了多个输入和输出的 shape，就需要显式地指定节点的下标，来表明取的是哪个输入。

```
    a = Input(shape=(3, 32, 32))
    b = Input(shape=(3, 64, 64))

    conv = Conv2D(16, (3, 3), padding='same')
    conved_a = conv(a)

    # Only one input so far, the following will work:
    assert conv.input_shape == (None, 3, 32, 32)

    conved_b = conv(b)
    # now the `.input_shape` property wouldn't work, but this
does:
    assert conv.get_input_shape_at(0) == (None, 3, 32, 32)
    assert conv.get_input_shape_at(1) == (None, 3, 64, 64)
```

第 5 章

Keras 网络结构

上一章我们掌握了 Keras 的 Sequential、函数式模型、常用的 API 功能及相关参数。本章进一步深入 Keras 的内部结构，详细介绍 Keras 的网络结构及其层的定义，将对每层的参数进行说明和分析。

5.1 Keras 层对象方法

Keras 网络中层对象有相同的方法。

- layer.get_weights()：用于返回层的权重（Numpy array）。
- layer.set_weights(weights)：用于从 Numpy array 中将权重加载到该层中，要求 Numpy array 的形状与 layer.get_weights()的形状相同。
- layer.get_config()：用于返回当前层配置信息的字典，层也可以由配置信息重构。

如果层仅有一个计算节点（即该层不是共享层），则可以通过下列方

法获得输入张量、输出张量、输入数据的形状和输出数据的形状。

- layer.input。
- layer.output。
- layer.input_shape。
- layer.output_shape。

如果该层有多个计算节点，可以使用下面的方法：

- layer.get_input_at(node_index)。
- layer.get_output_at(node_index)。
- layer.get_input_shape_at(node_index)。
- layer.get_output_shape_at(node_index)。

5.2 常用层

Keras 常用的网络层对象包括全连接层和激活层等。

5.2.1 Dense 层

Dense 是常用的全连接层，所实现的运算是 output = activation (dot(input, kernel)+bias)。其中，activation 是逐元素计算的激活函数；kernel 是本层的权值矩阵；bias 为偏置向量，只有当 use_bias=True 时，bias 偏置才生效。

```
    keras.layers.core.Dense(units, activation=None, use_bias=
True, kernel_initializer='glorot_uniform', bias_initializer=
'zeros',    kernel_regularizer=None,    bias_regularizer=None,
activity_  regularizer=None,  kernel_constraint=None,  bias_
constraint = None)
```

如果本层的输入数据的维度大于 2，则会先被压缩为与 kernel 相匹配的大小。

一个使用示例如下：

```
    # as first layer in a sequential model:
    # as first layer in a sequential model:
    model = Sequential()
    model.add(Dense(32, input_shape=(16,)))
    # now the model will take as input arrays of shape (*, 16)
    # and output arrays of shape (*, 32)

    # after the first layer, you don't need to specify
    # the size of the input anymore:
    model.add(Dense(32))
```

（1）参数。

- units：大于 0 的整数，代表该层的输出维度。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。

- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入。

形如(nb_samples, ..., input_shape[1])的 n 维张量，最常见的情况为(nb_samples, input_dim)的 2D 张量。

（3）输出。

形如(nb_samples, ..., units)的 n 维张量，最常见的情况为(nb_samples, output_dim)的 2D 张量。

5.2.2 Activation 层

激活层对一个层的输出施加激活函数。

```
keras.layers.core.Activation(activation)
```

（1）参数。

activation：将要使用的激活函数，为预定义激活函数名或一个

Tensorflow/Theano 的函数。参考激活函数。

（2）输入：shape。

任意，当使用激活层作为第一层时，要指定 input_shape。

（3）输出：shape。

与输入 shape 相同。

5.2.3　Dropout 层

为输入数据施加 Dropout。Dropout 将在训练过程中每次更新参数时，随机断开一定百分比（rate）的输入神经元。Dropout 层用于防止过拟合。

```
    keras.layers.core.Dropout(rate, noise_shape=None, seed =
None)
```

Dropout 层的参数如下。

- rate：0~1 的浮点数，用于控制需要断开的神经元的比例。
- noise_shape：整数张量，为将要应用在输入上的二值 Dropout mask 的 shape，例如输入为(batch_size, timesteps, features)，并且要求在各个时间步上的 Dropout mask 都相同，则可传入 noise_shape=(batch_size, 1, features)。
- seed：整数，使用的随机数种子。

5.2.4　Flatten 层

Flatten 层用来将输入“压平”，即把多维的输入一维化，常用在从卷

积层到全连接层的过渡。Flatten 不影响 batch 的大小。

```
keras.layers.core.Flatten()
```

一个使用示例如下：

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
                        border_mode='same',
                        input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

5.2.5 Reshape 层

Reshape 层用来将输入 shape 转换为特定的 shape。

```
keras.layers.core.Reshape(target_shape)
```

（1）参数。

target_shape：目标 shape，为整数的 tuple，不包含样本数目的维度（batch 大小）。

（2）输入 shape。

任意，但输入的 shape 必须固定。当使用该层为模型首层时，需要指定 input_shape 参数。

（3）输出 shape。

(batch_size,)+target_shape;

一个使用示例如下：

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

5.2.6　Permute 层

Permute层将输入的维度按照给定模式进行重排，例如，当需要将RNN和CNN网络连接时，可能会用到该层。

```
keras.layers.core.Permute(dims)
```

（1）参数。

dims：整数 tuple，指定重排的模式，不包含样本数的维度。重排模式的下标从 1 开始。例如，（2,1）代表将输入的第二个维度重排到输出的第一个维度，而将输入的第一个维度重排到第二个维度。

（2）输入 shape。

任意，当使用激活层作为第一层时，要指定 input_shape。

（3）输出 shape。

与输入相同，但是其维度按照指定的模式重新排列。

一个使用示例如下：

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
print model.output_shape
```

输出：

```
(None, 64, 10) # note: `None` is the batch dimension
```

5.2.7 RepeatVector 层

RepeatVector 层将输入重复 n 次。

```
keras.layers.core.RepeatVector(n)
```

（1）参数。

n：整数，表示重复的次数。

（2）输入 shape。

形如（nb_samples, features）的 2D 张量。

（3）输出 shape。

形如（nb_samples, n, features）的 3D 张量。

一个使用示例如下：

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension
```

```
model.add(RepeatVector(3))
# now: model.output_shape == (None, 3, 32)
```

5.2.8　Lambda 层

本函数用于对上一层的输出施以任何 Theano/Tensorflow 表达式。

```
keras.layers.core.Lambda(function, output_shape=None,
mask=None, arguments=None)
```

（1）参数。

- function：要实现的函数，该函数仅接受一个变量，即上一层的输出。
- output_shape：函数应该返回的值的 shape，既可以是一个 tuple，也可以是一个根据输入 shape 计算输出 shape 的函数。
- mask：掩码。
- arguments：可选，字典，用来记录向函数中传递的其他关键字参数。

（2）输入：shape。

任意，当使用该层作为第一层时，要指定 input_shape。

（3）输出：shape。

由 output_shape 参数指定的输出 shape，当使用 Tensorflow 时可自动推断。

一个使用示例如下：

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
          output_shape=antirectifier_output_shape))
```

5.2.9 ActivityRegularizer 层

经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值。

```
eras.layers.core.ActivityRegularization(l1=0.0, l2=0.0)
```

（1）参数。

- l1：表示 1 范数正则因子（正浮点数）。
- l2：表示 2 范数正则因子（正浮点数）。

（2）输入 shape。

任意，当使用该层作为第一层时，要指定 input_shape。

（3）输出 shape。

与输入 shape 相同。

5.2.10　Masking 层

使用给定的值对输入的序列信号进行“屏蔽”，用于定位需要跳过的时间步。

```
keras.layers.core.Masking(mask_value=0.0)
```

对于输入张量的时间步，即输入张量的第 1 维度（维度从 0 开始算，见下面的示例）。如果输入张量在该时间步上都等于 mask_value，则该时间步将在模型接下来的所有层（只要支持 masking）被跳过（屏蔽）。

如果模型接下来的一些层不支持 masking，却接收到 masking 过的数据，则抛出异常。

一个使用示例如下：

考虑输入数据 x 是一个形如(samples,timesteps,features)的张量，现将其送入 LSTM 层。因为缺少时间步为 3 和 5 的信号，所以将其屏蔽。这时候应该：

- 赋值 x[:,3,:] = 0，x[:,5,:] = 0。
- 在 LSTM 层之前插入 mask_value=0.的 Masking 层。

```
    model = Sequential()
    model.add(Masking(mask_value=0.,  input_shape=(timesteps,
features)))
    model.add(LSTM(32))
```

5.3 卷积层

5.3.1 Conv1D 层

一维卷积层（即时域卷积），用于在一维输入信号上进行邻域滤波。当使用该层作为首层时，需要提供关键字参数 input_shape。例如，(10,128)代表一个长为 10 的序列，序列中每个信号为 128 向量。而(None, 128)代表变长的 128 维向量序列。

该层生成将输入信号与卷积核按照单一的空域（或时域）方向进行卷积。如果 use_bias=True，则还会加上一个偏置项；若 activation 不为 None，则输出为经过激活函数的输出。

```
    keras.layers.convolutional.Conv1D(filters,  kernel_size,
strides=1, padding='valid', dilation_rate=1, activation=None,
use_bias=True,  kernel_initializer='glorot_uniform',  bias_
initializer='zeros', kernel_regularizer=None, bias_regularizer
=None, activity_regularizer=None, kernel_constraint=None, bias_
constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：整数或由单个整数构成的 list/tuple，卷积核的空域或时域窗长度。

- strides：整数或由单个整数构成的 list/tuple，为卷积的步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。
- padding：补 0 策略，为“valid”“same”或“causal”，“causal”将产生因果（膨胀的）卷积，即 output[t]不依赖于 input[t+1：]。当对不能违反时间顺序的时序信号建模时有用。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- dilation_rate：整数或由单个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers；
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。

- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入 shape。

形如（samples，steps，input_dim）的 3D 张量。

（3）输出 shape。

形如（samples，new_steps，nb_filter）的 3D 张量，因为有向量填充的原因，steps 的值会改变。

5.3.2 Conv2D 层

二维卷积层，即对图像的空域卷积。该层对二维输入进行滑动窗卷积，当使用该层作为第一层时，应提供 input_shape 参数。例如，input_shape = (128,128,3)代表 128*128 的彩色 RGB 图像（data_format='channels_last'）。

```
    keras.layers.convolutional.Conv2D(filters,    kernel_size,
strides=(1,  1),  padding='valid',  data_format=None,  dilation_
rate=(1, 1), activation=None, use_bias=True, kernel_initializer
='glorot_uniform',      bias_initializer='zeros',      kernel_
regularizer=None, bias_regularizer=None, activity_regularizer=
None, kernel_constraint=None, bias_constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：单个整数或由两个整数构成的 list/tuple，卷积核的

宽度和长度。如果为单个整数，则表示在各个空间维度的相同长度。

- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。
- padding：补 0 策略，为“valid”或“same”。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，则将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- dilation_rate：单个整数或由两个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入 shape。

“channels_first”模式下，输入形如（samples,channels，rows，cols）的 4D 张量。

“channels_last”模式下，输入形如（samples，rows，cols，channels）的 4D 张量。

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 input_shape。

（3）输出 shape。

“channels_first”模式下，为形如（samples，nb_filter，new_rows，new_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，new_rows，new_cols，nb_filter）的 4D 张量。

输出的行列数可能会因为填充方法而改变。

5.3.3　SeparableConv2D 层

该层是在深度方向上的可分离卷积。

可分离卷积[1]首先按深度方向进行卷积（对每个输入通道分别卷积），然后逐点进行卷积，将上一步的卷积结果混合到输出通道中。参数 depth_multiplier 控制了在 depthwise 卷积（第一步）的过程中，每个输入通道信号产生多少个输出通道，如图 5-1 所示。

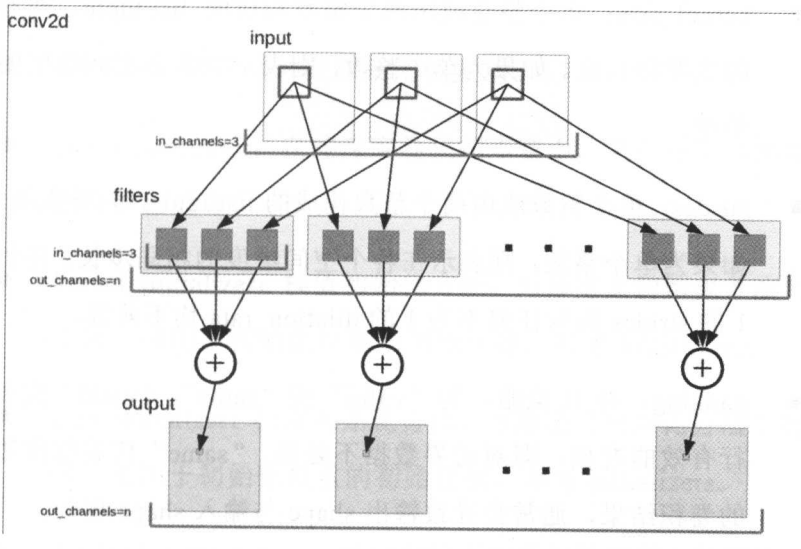


图 5-1　可分离卷积原理

直观来说，可分离卷积可以看做讲一个卷积核分解为两个小的卷积核，或看作 Inception 模块的一种极端情况。

当使用该层作为第一层时，应提供 input_shape 参数。例如 input_shape=（3,128,128）代表 128×128 的彩色 RGB 图像。

```
    keras.layers.convolutional.SeparableConv2D(filters,
kernel_size, strides=(1, 1), padding='valid', data_format=None,
depth_multiplier=1, activation=None, use_bias=True, depthwise_i
nitializer='glorot_uniform',    pointwise_initializer='glorot_
uniform', bias_initializer='zeros', depthwise_regularizer=None,
pointwise_regularizer=None,  bias_regularizer=None,  activity_
regularizer=None,    depthwise_constraint=None,    pointwise_
constraint=None, bias_constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：单个整数或由两个整数构成的 list/tuple，即卷积核的宽度和长度。如果为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如果为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。
- padding：补 0 策略，为“valid”或“same”。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），

或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。

- dilation_rate：单个整数或由两个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 ~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。
- use_bias：布尔值，是否使用偏置项。
- depth_multiplier：在按深度卷积的步骤中，每个输入通道使用多少个输出通道。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- depthwise_regularizer：施加在按深度卷积的权重上的正则项，为

Regularizer 对象。

- pointwise_regularizer：施加在按点卷积的权重上的正则项，为 Regularizer 对象。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。
- depthwise_constraint：施加在按深度卷积权重上的约束项，为 Constraints 对象。
- pointwise_constraint：施加在按点卷积权重的约束项，为 Constraints 对象。

（2）输入 shape。

“channels_first”模式下，输入形如（samples,channels，rows，cols）的 4D 张量。

“channels_last”模式下，输入形如（samples，rows，cols，channels）的 4D 张量。

（3）输出 shape

“channels_first”模式下，为形如（samples，nb_filter，new_rows，new_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，new_rows，new_cols，nb_filter）的 4D 张量。

输出的行列数可能会因为填充方法而改变。

5.3.4　Conv2DTranspose 层

该层是转置的卷积操作（反卷积）。需要反卷积的情况通常发生在用户想要对一个普通卷积的结果做反方向的变换。例如，将具有该卷积层输出 shape 的 tensor 还原为具有该卷积层输入 shape 的 tensor。同时保留与卷积层兼容的连接模式。此方法在生成对抗网络（GAN）中可能用到。

当使用该层作为第一层时，应提供 input_shape 参数。例如，input_shape = (3,128,128)代表 128×28 的彩色 RGB 图像。

```
    keras.layers.convolutional.Conv2DTranspose(filters,
kernel_size, strides=(1, 1), padding='valid', data_format=None,
activation=None,  use_bias=True,  kernel_initializer='glorot_
uniform',  bias_initializer='zeros',  kernel_regularizer=None,
bias_regularizer=None,    activity_regularizer=None,    kernel_
constraint=None, bias_constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：单个整数或由两个个整数构成的 list/tuple，即卷积核的宽度和长度。如果为单个整数，则表示在各个空间维度的相

同长度。

- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如果为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。
- padding：补 0 策略，为“valid”或“same”。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- dilation_rate：单个整数或由两个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 ~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入 shape。

“channels_first”模式下，输入形如（samples,channels，rows，cols）的 4D 张量。

“channels_last”模式下，输入形如（samples，rows，cols，channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples，nb_filter，new_rows，new_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，new_rows，new_cols，nb_filter）的 4D 张量。

输出的行列数可能会因为填充方法而改变。

5.3.5 Conv3D 层

三维卷积对三维的输入进行滑动窗卷积，当使用该层作为第一层时，应提供 input_shape 参数。例如 input_shape = (3,10,128,128)代表对 10 帧 128×128 的彩色 RGB 图像进行卷积。数据的通道位置仍然有 data_format 参数指定。

```
keras.layers.convolutional.Conv3D(filters, kernel_size, strides=(1, 1, 1), padding='valid', data_format=None, dilation_rate=(1, 1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：单个整数或由 3 个整数构成的 list/tuple，即卷积核的宽度和长度。如果为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由 3 个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。

- padding：补 0 策略，为“valid”或“same”。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- dilation_rate：单个整数或由 3 个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。

- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入：shape。

“channels_first”模式下，输入应为形如（samples，channels，input_dim1，input_dim2, input_dim3）的 5D 张量。

“channels_last”模式下，输入应为形如（samples，input_dim1，input_dim2, input_dim3，channels）的 5D 张量。

（2）输出：shape。

“channels_first”模式下，输入应为形如（samples，filters，new_conv_dim1, new_conv_dim2, new_conv_dim3）的 5D 张量。

“channels_last”模式下，输入应为形如（samples，new_conv_dim1，new_conv_dim2, new_conv_dim3, filters）的 5D 张量。

new_conv_dim1, new_conv_dim2 and new_conv_dim3 可能会因为填充方法而改变。

5.3.6　Cropping1D 层

在时间轴（axis1）上对 1D 输入（即时间序列）进行裁剪。

```
keras.layers.convolutional.Cropping1D(cropping=(1, 1))
```

（1）参数。

cropping：长为 2 的 tuple，指定在序列的首尾要裁剪掉多少个元素。

（2）输入 shape。

形如（samples，axis_to_crop，features）的 3D 张量。

（3）输出 shape。

形如（samples，cropped_axis，features）的 3D 张量。

5.3.7　Cropping2D 层

对 2D 输入（图像）进行裁剪，将在空域维度，即宽和高的方向上的裁剪。

```
keras.layers.convolutional.Cropping2D(cropping=((0, 0), (0,
0)), data_format=None)
```

（1）参数。

- cropping：长为 2 的整数 tuple，分别为宽和高的方向上的头部与*尾部需要裁剪掉的元素数*。

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，

“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入：shape。

“channels_first”模式下，输入应为形如（batch, channels, rows, cols）的 4D 张量。

“channels_last”模式下，输入应为形如（batch, rows, cols, channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，输入应为形如（batch, channels, cropped_rows, cropped_cols）的 4D 张量。

“channels_last”模式下，输入应为形如（batch, cropped_rows, cropped_cols, channels）的 4D 张量。

5.3.8 Cropping3D 层

对 3D 输入（图像）进行裁剪。

```
    keras.layers.convolutional.Cropping3D(cropping=((1, 1), (1,
1), (1, 1)), data_format=None)
```

（1）参数。

- cropping：长为 3 的整数 tuple，分别为 3 个方向的头部与尾部需

要裁剪掉的元素数。

- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入：shape。

形如（samples, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop）的 5D 张量。

（3）输出：shape。

形如（samples, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis）的 5D 张量。

5.3.9　UpSampling1D 层

对 1D 输入数据的上采样层。

在时间轴上，将每个时间步重复 length 次。

```
keras.layers.convolutional.UpSampling1D(size=2)
```

（1）参数。

size：上采样因子。

（2）输入 shape。

形如（samples，steps，features）的 3D 张量。

（3）输出 shape。

形如（samples，upsampled_steps，features）的 3D 张量。

5.3.10 UpSampling2D 层

对 2D 输入数据的上采样层。

将数据的行和列分别重复 size[0]和 size[1]次。

```
    keras.layers.convolutional.UpSampling2D(size=(2, 2), data_
format=None)
```

（1）参数。

- size：整数 tuple，分别为行和列上采样因子。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”字段，“channels_first”对应原本的“th”字段。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples，channels, rows，cols）的

4D 张量。

“channels_last”模式下，为形如（samples，rows, cols，channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples，channels, upsampled_rows, upsampled_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，upsampled_rows, upsampled_cols，channels）的 4D 张量。

5.3.11　UpSampling3D 层

对 3D 输入数据的上采样层。

将数据的三个维度上分别重复 size[0]、size[1]和 ize[2]次。

```
    keras.layers.convolutional.UpSampling3D(size=(2,  2,  2),
data_format=None)
```

本层目前只能在使用 Theano 为后端时可用。

（1）参数。

- size：长为 3 的整数 tuple，代表在 3 个维度上的上采样因子。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”

应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples, channels, dim1, dim2, dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, upsampled_dim1, upsampled_dim2, upsampled_dim3,channels,）的 5D 张量。

5.3.12 ZeroPadding1D 层

对 1D 输入的首尾端（如时域序列）填充 0，以控制卷积以后向量的长度。

```
keras.layers.convolutional.ZeroPadding1D(padding=1)
```

（1）参数。

- padding：整数，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 1（第 1 维，第 0 维是样本数）。

（2）输入 shape。

形如（samples，axis_to_pad，features）的 3D 张量。

（3）输出 shape。

形如（samples，paded_axis，features）的 3D 张量。

5.3.13　ZeroPadding2D 层

对 2D 输入（如图片）的边界填充 0，以控制卷积以后特征图的大小。

```
    keras.layers.convolutional.ZeroPadding2D(padding=(1,   1),
data_format=None)
```

（1）参数。

- padding：整数 tuple，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 3 和轴 4（即在'th'模式下图像的行和列，在“channels_last”模式下要填充的则是轴 2，3）。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，形如（samples，channels，first_axis_to_pad，second_axis_to_pad）的 4D 张量。

“channels_last”模式下，形如（samples，first_axis_to_pad，second_axis_to_pad, channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，形如（samples，channels，first_paded_axis，second_paded_axis）的 4D 张量。

“channels_last”模式下，形如（samples，first_paded_axis，second_paded_axis, channels）的 4D 张量。

5.3.14 ZeroPadding3D 层

将数据的 3 个维度上填充 0。

```
    keras.layers.convolutional.ZeroPadding3D(padding=(1, 1, 1),
data_format=None)
```

本层目前只能在使用 Theano 为后端时可用。

（1）参数。

- padding：整数 tuple，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 3，轴 4 和轴 5，“channels_last”模式下则是轴 2，3 和 4。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的

image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples, channels, first_axis_to_pad, first_axis_to_pad, first_axis_to_pad,）的 5D 张量。

“channels_last”模式下，为形如（samples, first_axis_to_pad, first_axis_to_pad, first_axis_to_pad, channels）的 5D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples, channels, first_paded_axis, second_paded_axis, third_paded_axis,）的 5D 张量。

“channels_last”模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3, channels,）的 5D 张量。

5.4　池化层

5.4.1　MaxPooling1D 层

对时域 1D 信号进行最大值池化。

```
    keras.layers.pooling.MaxPooling1D(pool_size=2,
strides=None, padding='valid')
```

（1）参数。

- pool_size：整数，表示池化窗口大小。
- strides：整数或 None，下采样因子，例如，设 2 将会使得输出 shape 为输入的一半，若为 None，则默认值为 pool_size。
- padding：“valid”或者“same”。

（2）输入 shape。

形如（samples，steps，features）的 3D 张量。

（3）输出 shape。

形如（samples，downsampled_steps，features）的 3D 张量。

5.4.2 MaxPooling2D 层

为空域信号施加最大值池化。

```
    keras.layers.pooling.MaxPooling2D(pool_size=(2,       2),
strides=None, padding='valid', data_format=None)
```

（1）参数。

- pool_size：整数或长为 2 的整数 tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。该值为整数，意为各个维度值相同且为该数字；

- strides：整数或长为 2 的整数 tuple，或者 None，步长值。
- border_mode：“valid”或者“same”。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples，channels, rows，cols）的 4D 张量。

“channels_last”模式下，为形如（samples，rows, cols，channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples，channels, pooled_rows, pooled_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，pooled_rows, pooled_cols，channels）的 4D 张量。

5.4.3 MaxPooling3D 层

为 3D 信号（空域或时空域）施加最大值池化。

```
keras.layers.pooling.MaxPooling3D(pool_size=(2, 2, 2),
strides=None, padding='valid', data_format=None)
```

本层目前只能在使用 Theano 为后端时可用。

（1）参数。

- pool_size：整数或长为 3 的整数 tuple，代表在 3 个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为 3 的整数 tuple，或者 None，步长值。
- padding：“valid”或者“same”。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples, channels, len_pool_dim1,

len_pool_dim2, len_pool_dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的 5D 张量。

5.4.4　AveragePooling1D 层

对时域 1D 信号进行平均值池化。

```
    keras.layers.pooling.AveragePooling1D(pool_size=2,
strides=None, padding='valid')
```

（1）参数。

- pool_size：整数，表示池化窗口大小。
- strides：整数或 None，下采样因子，例如，设 2 将会使得输出 shape 为输入的一半，若为 None，则默认值为 pool_size。
- padding：“valid”或者“same”。

（2）输入 shape。

形如（samples，steps，features）的 3D 张量。

（3）输出 shape。

形如（samples，downsampled_steps，features）的 3D 张量。

5.4.5 AveragePooling2D 层

为空域信号施加平均值池化。

```
    keras.layers.pooling.AveragePooling2D(pool_size=(2,    2),
strides=None, padding='valid', data_format=None)
```

（1）参数。

- pool_size：整数或长为 2 的整数 tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。该值为整数，意为各个维度值相同且为该数字。
- strides：整数或长为 2 的整数 tuple，或者 None，步长值。
- border_mode：“valid”或者“same”。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples，channels, rows，cols）的 4D 张量。

“channels_last”模式下，为形如（samples，rows, cols，channels）的 4D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples，channels, pooled_rows, pooled_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，pooled_rows, pooled_cols，channels）的 4D 张量。

5.4.6　AveragePooling3D 层

为 3D 信号（空域或时空域）施加平均值池化。

```
    keras.layers.pooling.AveragePooling3D(pool_size=(2, 2, 2),
strides=None, padding='valid', data_format=None)
```

本层目前只能在使用 Theano 为后端时可用。

（1）参数。

- pool_size：整数或长为 3 的整数 tuple，代表在 3 个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为 3 的整数 tuple，或者 None，步长值。

- padding：“valid”或者“same”。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128×128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量。

（3）输出 shape。

“channels_first”模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的 5D 张量。

“channels_last”模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的 5D 张量。

5.4.7 GlobalMaxPooling1D 层

对于时间信号的全局最大池化。

```
keras.layers.pooling.GlobalMaxPooling1D()
```

（1）输入 shape。

形如（samples, steps, features）的 3D 张量。

（2）输出 shape。

形如（samples, features)的 2D 张量。

5.4.8　GlobalAveragePooling1D 层

为时域信号施加全局平均值池化。

```
keras.layers.pooling.GlobalAveragePooling1D()
```

（1）输入 shape。

形如（samples，steps，features）的 3D 张量。

（2）输出 shape。

形如（samples, features）的 2D 张量。

5.4.9　GlobalMaxPooling2D 层

为空域信号施加全局最大值池化。

```
keras.layers.pooling.GlobalMaxPooling2D(dim_ordering='default')
```

（1）参数。

- data_format：字符串，“channels_first”或“channels_last”之一，

代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples，channels, rows，cols）的 4D 张量。

“channels_last”模式下，为形如（samples，rows, cols，channels）的 4D 张量。

（3）输出 shape。

形如（nb_samples, channels）的 2D 张量。

5.4.10 GlobalAveragePooling2D 层

为空域信号施加全局平均值池化。

```
    keras.layers.pooling.GlobalAveragePooling2D(dim_ordering=
'default')
```

（1）参数。

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_

dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

（2）输入 shape。

“channels_first”模式下，为形如（samples，channels, rows，cols）的 4D 张量。

“channels_last”模式下，为形如（samples，rows, cols，channels）的 4D 张量。

（3）输出 shape。

形如（nb_samples, channels）的 2D 张量。

5.5　局部连接层

5.5.1　LocallyConnected1D 层

LocallyConnected1D 层与 Conv1D 工作方式类似，唯一的区别是不进行权值共享，即施加在不同输入位置的滤波器是不一样的。

```
    keras.layers.local.LocallyConnected1D(filters, kernel_size,
strides=1, padding='valid', data_format=None, activation=None,
use_bias=True,    kernel_initializer='glorot_uniform',    bias_
initializer='zeros', kernel_regularizer=None, bias_regularizer
=None,    activity_regularizer=None,    kernel_constraint=None,
```

```
bias_constraint=None)
```

（1）参数。

- filters：卷积核的数目（即输出的维度）。
- kernel_size：整数或由单个整数构成的 list/tuple，卷积核的空域或时域窗长度。
- strides：整数或由单个整数构成的 list/tuple，为卷积的步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容。
- padding：补 0 策略，目前仅支持 valid（大小写敏感）。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- dilation_rate：整数或由单个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。

- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入 shape。

形如（samples，steps，input_dim）的 3D 张量。

（3）输出 shape

形如（samples，new_steps，nb_filter）的 3D 张量，因为有向量填充的原因，steps 的值会改变。

5.5.2　LocallyConnected2D 层

LocallyConnected2D 层与 Convolution2D 工作方式类似，唯一的区别是不进行权值共享。即施加在不同输入 patch 的滤波器是不一样的，当使用该层作为模型首层时，需要提供参数 input_dim 或 input_shape 参数。参数含义参考 Convolution2D。

```
    keras.layers.local.LocallyConnected2D(filters, kernel_size,
strides=(1, 1), padding='valid', data_format=None, activation=
None,    use_bias=True,    kernel_initializer='glorot_uniform',
bias_initializer='zeros',    kernel_regularizer=None,    bias_
regularizer=None, activity_regularizer=None, kernel_constraint
=None, bias_constraint=None)
```

（1）参数。

- filters：表示卷积核的数目（即输出的维度）。
- kernel_size：单个整数或由两个整数构成的 list/tuple，即卷积核的宽度和长度。如果为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长；
- padding：补 0 策略，目前仅支持 valid（大小写敏感）。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128×128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。

- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。

（2）输入 shape。

“channels_first”模式下，输入形如（samples,channels，rows，cols）的 4D 张量。

“channels_last”模式下，输入形如（samples，rows，cols，channels）的 4D 张量。

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 input_shape。

（3）输出 shape。

“channels_first”模式下，为形如（samples，nb_filter，new_rows，new_cols）的 4D 张量。

“channels_last”模式下，为形如（samples，new_rows，new_cols，

nb_filter）的 4D 张量。

输出的行列数可能会因为填充方法而改变。

一个使用示例如下：

```
    # apply a 3x3 unshared weights convolution with 64 output
filters on a 32x32 image
    # with `data_format="channels_last"`:
    model = Sequential()
    model.add(LocallyConnected2D(64, (3, 3), input_shape=(32,
32, 3)))
    # now model.output_shape == (None, 30, 30, 64)
    # notice that this layer will consume (30*30)*(3*3*3*64) +
(30*30)*64 parameters

    # add a 3x3 unshared weights convolution on top, with 32 output
filters:
    model.add(LocallyConnected2D(32, (3, 3)))
    # now model.output_shape == (None, 28, 28, 32)
```

5.6 循环层

5.6.1 Recurrent 层

这是循环层的抽象类，请不要在实际模型中直接应用该层（因为它是抽象类，无法实例化任何对象）。请使用它的子类 LSTM、GRU 或 SimpleRNN。

所有的循环层（LSTM,GRU,SimpleRNN）都服从本层的性质，并接受本层指定的所有关键字参数。

```
    keras.layers.recurrent.Recurrent(return_sequences=False,
```

```
go_backwards=False,        stateful=False,        unroll=False,
implementation=0)
```

（1）参数。

- weights：Numpy array 的 List，用以初始化权重。该 List 形如 [(input_dim, output_dim), (output_dim, output_dim), (output_dim)]。
- return_sequences：布尔值，默认 False，控制返回类型。若为 True，则返回整个序列，否则，仅返回输出序列的最后一个输出。
- go_backwards：布尔值，默认为 False。若为 True，则逆向处理输入序列并返回逆序后的序列。
- stateful：布尔值，默认为 False。若为 True，则一个 batch 中下标为 i 的样本的最终状态将会用作下一个 batch 同样下标的样本的初始状态。
- unroll：布尔值，默认为 False。若为 True，则循环层将被展开，否则就使用符号化的循环。当使用 TensorFlow 为后端时，循环网络本来就是展开的，因此，该层不做任何事情。层展开会占用更多的内存，但会加速 RNN 的运算。层展开只适用于短序列。
- implementation：0，1 或 2，若为 0，则 RNN 将以更少但是更大的矩阵乘法实现，因此在 CPU 上运行更快，但消耗更多的内存。如果设为 1，则 RNN 将以更多但更小的矩阵乘法实现，因此在 CPU 上运行更慢，在 GPU 上运行更快，并且消耗更少的内存。如果设为 2（仅 LSTM 和 GRU 可以设为 2），则 RNN 将把输入

门、遗忘门和输出门合并为单个矩阵，以获得更加在 GPU 上更加高效的实现。注意，RNN dropout 必须在所有门上共享，并导致正则效果性能微弱降低。

- input_dim：输入维度，当使用该层为模型首层时，应指定该值（或等价的指定 input_shape）。
- input_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接 Flatten 层，然后又要连接 Dense 层时，需要指定该参数，否则全连接的输出无法计算出来。注意，如果循环层不是网络的第一层，则需要在网络的第一层中指定序列的长度（通过 input_shape 指定）。

（2）输入 shape。

形如（samples，timesteps，input_dim）的 3D 张量。

（3）输出 shape。

如果 return_sequences=True：返回形如（samples，timesteps，output_dim）的 3D 张量，否则，返回形如（samples，output_dim）的 2D 张量。

一个使用示例如下：

```
# as the first layer in a Sequential model
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# now model.output_shape == (None, 32)
# note: `None` is the batch dimension.
```

```
    # the following is identical:
    model = Sequential()
    model.add(LSTM(32, input_dim=64, input_length=10))

    # for subsequent layers, no need to specify the input size:
            model.add(LSTM(16))

    # to stack recurrent layers, you must use return_sequences
=True
    # on any recurrent layer that feeds into another recurrent
layer.
    # note that you only need to specify the input size on the
first layer.
    model = Sequential()
    model.add(LSTM(64, input_dim=64, input_length=10, return_
sequences=True))
    model.add(LSTM(32, return_sequences=True))
    model.add(LSTM(10))
```

RNN 的初始状态可以通过设置 initial_state 用符号式的方式指定 RNN 层的初始状态。即 initial_state 的值应该为一个 tensor 或一个 tensor 列表，代表 RNN 层的初始状态。

用户也可以通过设置 reset_states 参数用数值的方法设置 RNN 的初始状态，状态的值应该为 Numpy 数组或 Numpy 数组的列表，代表 RNN 层的初始状态。

循环层支持通过时间步变量对输入数据进行 Masking，如果想将输入数据的一部分屏蔽掉，请使用 Embedding 层并将参数 mask_zero 设为 True。

用户可以将 RNN 设置为“stateful”，意味着由每个 batch 计算出的状态都会被重用于初始化下一个 batch 的初始状态。状态 RNN 假设连续的

两个 batch 之中，相同下标的元素有一一映射关系。

要启用状态 RNN，请在实例化层对象时指定参数 stateful=True，并在 Sequential 模型中使用固定大小的 batch：通过在模型的第一层传入 batch_size=(...)和 input_shape 来实现。在函数式模型中，对所有的输入都要指定相同的 batch_size。

如果要将循环层的状态重置，请调用.reset_states()方法，对模型的调用将重置模型中所有状态 RNN 的状态。对单个层调用，则只重置该层的状态。

5.6.2 SimpleRNN 层

全连接 RNN 网络，RNN 的输出会被回馈到输入。

```
    keras.layers.recurrent.SimpleRNN(units, activation='tanh',
use_bias=True, kernel_initializer='glorot_uniform', recurrent_
initializer='orthogonal',  bias_initializer='zeros',  kernel_
regularizer=None, recurrent_regularizer=None, bias_regularizer
=None,   activity_regularizer=None,   kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0)
```

参数如下。

- units：输出维度。
- activation：激活函数，为预定义的激活函数名（参考激活函数）。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字

符串，或用于初始化权重的初始化器。参考 initializers。

- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。
- dropout：0～1 之间的浮点数，用于控制输入线性变换的神经元断开比例。
- recurrent_dropout：0～1 之间的浮点数，用于控制循环状态的线性变换的神经元断开比例。

5.6.3 GRU 层

门限循环单元。

```
keras.layers.recurrent.GRU(units, activation='tanh',
recurrent_activation='hard_sigmoid', use_bias=True, kernel_
initializer='glorot_uniform', recurrent_initializer=
'orthogonal', bias_initializer='zeros', kernel_regularizer=
None, recurrent_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, recurrent_
constraint=None, bias_constraint=None, dropout=0.0, recurrent_
dropout=0.0)
```

参数如下。

- units：输出维度。
- activation：激活函数，为预定义的激活函数名（参考激活函数）。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer

对象。

- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象。
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。
- dropout：0～1 之间的浮点数，用于控制输入线性变换的神经元断开比例。
- recurrent_dropout：0～1 之间的浮点数，用于控制循环状态的线性变换的神经元断开比例。

5.6.4　LSTM 层

Keras 的长短期记忆模型。

```
    keras.layers.recurrent.LSTM(units,      activation='tanh',
recurrent_activation='hard_sigmoid',  use_bias=True,  kernel_
initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros',    unit_forget_bias=True,    kernel_
regularizer=None, recurrent_regularizer=None, bias_regularizer
=None,   activity_regularizer=None,   kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0)
```

参数如下。

- units：输出维度。
- activation：激活函数，为预定义的激活函数名（参考激活函数）。
- recurrent_activation：为循环步施加的激活函数（参考激活函数）。
- use_bias：布尔值，是否使用偏置项。
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象。
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象。
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象。
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象。
- recurrent_constraints：施加在循环核上的约束项，为 Constraints

对象。

- bias_constraints：施加在偏置上的约束项，为 Constraints 对象。
- dropout：0～1 之间的浮点数，用于控制输入线性变换的神经元断开比例。
- recurrent_dropout：0～1 之间的浮点数，用于控制循环状态的线性变换的神经元断开比例。

5.7　嵌入层

嵌入层将正整数（下标）转换为具有固定大小的向量，如[[4],[20]]->[[0.25,0.1], [0.6,-0.2]]，Embedding 层只能作为模型的第一层。

```
    keras.layers.embeddings.Embedding(input_dim,  output_dim,
embeddings_initializer='uniform', embeddings_regularizer=None,
activity_regularizer=None,  embeddings_constraint=None,  mask_
zero=False, input_length=None)
```

（1）参数。

- input_dim：大或等于 0 的整数，字典长度，即输入数据最大下标+1。
- output_dim：大于 0 的整数，代表全连接嵌入的维度。
- embeddings_initializer：嵌入矩阵的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers。

- embeddings_regularizer：嵌入矩阵的正则项，为 Regularizer 对象。
- embeddings_constraint：嵌入矩阵的约束项，为 Constraints 对象。
- mask_zero：布尔值，确定是否将输入中的“0”看作是应该被忽略的“填充”（padding）值，该参数在使用递归层处理变长输入时有用。设置为 True 的话，则模型中后续的层必须都支持 masking，否则会抛出异常。如果该值为 True，则下标 0 在字典中不可用，input_dim 应设置为|vocabulary| + 2。
- input_length：当输入序列的长度固定时，该值为其长度。如果要在该层后接 Flatten 层，然后接 Dense 层，则必须指定该参数，否则 Dense 层的输出维度无法自动推断。

（2）输入 shape。

形如（samples，sequence_length）的 2D 张量。

（3）输出 shape。

形如(samples, sequence_length, output_dim)的 3D 张量。

一个使用示例如下：

```
    model = Sequential()
    model.add(Embedding(1000, 64, input_length=10))
    # the model will take as input an integer matrix of size (batch,
input_length).
    # the largest integer (i.e. word index) in the input should
be no larger than 999 (vocabulary size).
    # now model.output_shape == (None, 10, 64), where None is the
batch dimension.
```

```
input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

5.8　融合层

Merge 层提供了一系列用于融合两个层或两个张量的层对象和方法。以大写首字母开头的是 Layer 类，以小写字母开头的是张量的函数。小写字母开头的张量函数在内部实际上是调用了大写字母开头的层。

（1）Add：该层接收一个列表的同 shape 张量，并返回它们的和，shape 不变。

```
keras.layers.merge.Add()
```

（2）Multiply：该层接收一个列表的同 shape 张量，并返回它们的逐元素积的张量，shape 不变。

```
keras.layers.merge.Multiply()
```

（3）Average：该层接收一个列表的同 shape 张量，并返回它们的逐元素均值，shape 不变。

```
keras.layers.merge.Average()
```

（4）Maximum：该层接收一个列表的同 shape 张量，并返回它们的逐元素最大值，shape 不变。

```
keras.layers.merge.Maximum()
```

（5）Concatenate：该层接收一个列表的同 shape 张量，并返回它们的

按照给定轴相接构成的向量。

```
keras.layers.merge.Concatenate(axis=-1)
```

参数 axis：表示连接的轴。

（6）Dot：计算两个 tensor 中样本的张量乘积。例如，如果两个张量 a 和 b 的 shape 都为（batch_size, n），则输出为形如（batch_size,1）的张量，结果张量每个 batch 的数据都是 a[i,:]和 b[i,:]的矩阵（向量）点积。

```
keras.layers.merge.Dot(axes, normalize=False)
```

参数如下。

- axes：整数或整数的 tuple，执行乘法的轴。
- normalize：布尔值，是否沿执行成绩的轴做 L2 规范化，如果设为 True，那么乘积的输出是两个样本的余弦相似性。

（7）add：Add 层的函数式包装。

```
add(inputs)
```

参数 inputs：长度至少为 2 的张量列表。

返回值：输入列表张量之和。

（8）multiply：Multiply 的函数包装。

```
multiply(inputs)
```

参数 inputs：长度至少为 2 的张量列表。

返回值：输入列表张量之逐元素积

（9）average：求平均的函数包装。

```
average(inputs)
```

参数 inputs：长度至少为 2 的张量列表。

返回值：输入列表张量之逐元素均值

（10）maximum：求最大值的函数包装。

```
maximum(inputs)
```

参数 inputs：长度至少为 2 的张量列表。

返回值：输入列表张量之逐元素均值。

（11）concatenate：合并值的函数包装。

```
concatenate(inputs, axis=-1))
```

参数如下。

- inputs：长度至少为 2 的张量列表。
- axis：相接的轴。

（12）dot：Dot 的函数包装。

```
dot(inputs, axes, normalize=False)
```

参数如下。

- inputs：长度至少为 2 的张量列表。
- axes：整数或整数的 tuple，执行乘法的轴。

- normalize：布尔值，是否沿执行成绩的轴做 L2 规范化，如果设为 True，那么乘积的输出是两个样本的余弦相似性。

5.9 激活层

5.9.1 LeakyReLU 层

LeakyRelU 是修正线性单元（Rectified Linear Unit，ReLU）的特殊版本，当不激活时，LeakyReLU 仍然会有非零输出值，从而获得一个小梯度，避免 ReLU 可能出现的神经元“死亡”现象。即 f(x)=alpha * x for x < 0，f(x) = x for x>=0。

```
keras.layers.advanced_activations.LeakyReLU(alpha=0.3)
```

（1）参数。

alpha：大于 0 的浮点数，代表激活函数图像中第三象限线段的斜率。

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.9.2 PReLU 层

该层为参数化的 ReLU（Parametric ReLU），表达式是：f(x) = alpha * x for x < 0，f(x) = x for x>=0，此处的 alpha 为一个与 xshape 相同的可学

习的参数向量。

```
    keras.layers.advanced_activations.PReLU(alpha_initializer
='zeros',    alpha_regularizer=None,    alpha_constraint=None,
shared_axes=None)
```

（1）参数。

- alpha_initializer：alpha 的初始化函数。
- alpha_regularizer：alpha 的正则项。
- alpha_constraint：alpha 的约束项。
- shared_axes：该参数指定的轴将共享同一组科学系参数，例如假如输入特征图是从 2D 卷积过来的，具有形如(batch, height, width, channels)这样的 shape。如果希望在空域共享参数，这样每个 filter 就只有一组参数，则设定 shared_axes=[1,2]可完成该目标。

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.9.3　ELU 层

ELU 层是指数线性单元（Exponential Linera Unit），表达式为：该层为参数化的 ReLU(Parametric ReLU)，表达式是：f(x) = alpha * (exp(x) - 1.) for x < 0，f(x) = x for x>=0。

```
keras.layers.advanced_activations.ELU(alpha=1.0)
```

（1）参数。

alpha：控制负因子的参数；

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.9.4 ThresholdedReLU 层

该层是带有门限的 ReLU，表达式是：f(x) = x for x > theta，f(x) = 0 otherwise。

```
keras.layers.advanced_activations.ThresholdedReLU(theta=
1.0)
```

（1）参数。

theata：大或等于 0 的浮点数，激活门限位置。

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.10　规范层

该层在每个 batch 上将前一层的激活值重新规范化，即使得其输出数据的均值接近 0，其标准差接近 1。

```
    keras.layers.normalization.BatchNormalization(axis=-1,
momentum=0.99, epsilon=0.001, center=True, scale=True, beta_
initializer='zeros', gamma_initializer='ones', moving_mean_
initializer='zeros', moving_variance_initializer='ones', beta_
regularizer=None, gamma_regularizer=None, beta_constraint=None,
gamma_constraint=None)
```

（1）参数。

- axis：整数，指定要规范化的轴，通常为特征轴。例如在进行 data_format="channels_first 的 2D 卷积后，一般会设 axis=1。
- momentum：动态均值的动量。
- epsilon：大于 0 的小浮点数，用于防止除 0 错误。
- center：若设为 True，将会将 beta 作为偏置加上去，否则忽略参数 beta。
- scale：若设为 True，则会乘以 gamma，否则不使用 gamma。当下一层是线性的时，可以设 False，因为 scaling 的操作将被下一层执行。
- beta_initializer：beta 权重的初始方法。
- gamma_initializer：gamma 的初始化方法。

- moving_mean_initializer：动态均值的初始化方法。
- moving_variance_initializer：动态方差的初始化方法。
- beta_regularizer：可选的 beta 正则。
- gamma_regularizer：可选的 gamma 正则。
- beta_constraint：可选的 beta 约束。
- gamma_constraint：可选的 gamma 约束。

（2）输入 shape。

任意，当使用本层为模型首层时，指定 input_shape 参数时有意义。

（3）输出 shape。

与输入 shape 相同。

BN 层的作用如下：

（1）加速收敛。

（2）控制过拟合，可以少用或不用 Dropout 和正则。

（3）降低网络对初始化权重不敏感。

（4）允许使用较大的学习率。

5.11　噪声层

5.11.1　GaussianNoise 层

该层为数据施加 0 均值，标准差为 stddev 的加性高斯噪声。该层在克服过拟合时比较有用，可以将它看作是随机的数据提升。高斯噪声是需要对输入数据进行破坏时的自然选择。

一个使用噪声层的典型案例是构建去噪自动编码器，即 Denoising AutoEncoder（DAE）。该编码器试图从加噪的输入中重构无噪信号，以学习到原始信号的鲁棒性表示。

因为这是一个起正则化作用的层，该层只在训练时才有效。

```
keras.layers.noise.GaussianNoise(stddev)
```

（1）参数。

stddev：浮点数，代表要产生的高斯噪声标准差。

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.11.2　GaussianDropout 层

该层为输入施加以 1 为均值，标准差为 sqrt(rate/(1-rate)的乘性高斯噪

声，因为这是一个起正则化作用的层，该层只在训练时才有效。

```
keras.layers.noise.GaussianDropout(rate)
```

（1）参数。

rate：浮点数，断连概率，与 Dropout 层相同。

（2）输入 shape。

任意，当使用该层为模型首层时需指定 input_shape 参数。

（3）输出 shape。

与输入相同。

5.12 包装器 Wrapper

5.12.1 TimeDistributed 层

该包装器可以把一个层应用到输入的每一个时间步上。

```
keras.layers.wrappers.TimeDistributed(layer)
```

参数 layer：Keras 层对象。

输入至少为 3D 张量，下标为 1 的维度将被认为是时间维。

例如，考虑一个含有 32 个样本的 batch，每个样本都是 10 个向量组成的序列，每个向量长为 16，则其输入维度为（2,10,16），其不包含 batch 大小的 input_shape 为（10,16）。

用户可使用包装器 TimeDistributed 包装 Dense，以产生针对各个时间

步信号的独立全连接。

```
    # as the first layer in a model
    model = Sequential()
    model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
    # now model.output_shape == (None, 10, 8)

    # subsequent layers: no need for input_shape
    model.add(TimeDistributed(Dense(32)))
    # now model.output_shape == (None, 10, 32)
```

程序的输出数据 shape 为（32,10,8）。

使用 TimeDistributed 包装 Dense 严格等价于 layers.TimeDistribuedDense。不同的是，包装器 TimeDistribued 还可以对其他层进行包装，如这里对 Convolution2D 包装。

```
    model = Sequential()
    model.add(TimeDistributed(Convolution2D(64, 3, 3), input_
shape=(10, 3, 299, 299)))
```

5.12.2　Bidirectional 层

双向 RNN 包装器。

```
    keras.layers.wrappers.Bidirectional(layer,
merge_mode='concat', weights=None)
```

参数如下。

- layer：Recurrent 对象。
- merge_mode：前向和后向 RNN 输出的结合方式，为 sum、mul、concat、ave 和 None 之一。若设为 None，则返回值不结合，而

是以列表的形式返回。

一个使用示例如下：

```
    model = Sequential()
    model.add(Bidirectional(LSTM(10,   return_sequences=True),
input_shape=(5, 10)))
    model.add(Bidirectional(LSTM(10)))
    model.add(Dense(5))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy',
optimizer='rmsprop')
```

5.13 自定义层

对于简单的定制操作，可以通过使用 layers.core.Lambda 层来完成。但对于任何具有可训练权重的定制层，应该自定义实现。

Keras2 的自定义层提供定制层的 API，需要实现下面三个方法：

- build(input_shape)：这是定义权重的方法，可训练的权应该在这里被加入列表 self.trainable_weights 中。其他的属性包括 self.non_trainabe_weights（列表）和 self.updates（需要更新的形如（tensor, new_tensor）的 tuple 的列表）。可参考 BatchNormalization 层的实现来学习如何使用上面的两个属性。这个方法必须设置 self.built = True，可通过调用 super([layer],self).build()来实现。
- call(x)：这是定义层功能的方法，只需关注第一个参数，即输入张量。

- compute_output_shape(input_shape)：如果层修改了输入数据的 shape，则应该指定 shape 变化的方法。这个函数使得 Keras 可以做自动 shape 推断。

```
from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

class MyLayer(Layer):

    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(shape=(input_shape[1],
self.output_dim),
                                      initializer='uniform',
                                      trainable=True)
        super(MyLayer, self).build(input_shape)  # Be sure
to call this somewhere!

    def call(self, x):
        return K.dot(x, self.kernel)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

参考文献

[1] http://qiita.com/YusukeSuzuki@github/items/0764d15b9d0b97ec1e16

第 6 章

Keras 数据预处理

上一章我们详细介绍了 Keras 的网络结构以及层的定义，并对常用层的参数进行了详解，从而使我们知晓了 Keras 功能的丰富性，通过组合不同的层就能快速满足深度学习的应用。Keras 除了拥有丰富的参数和 API，还支持数据预处理功能。早在 2015 年前发布 0.1.0 版，Keras 就专为数据预处理写了一个独立模块，代码位于 keras.preprocessing 下。这一点更明确了 Keras 最初的定位：一个高层的深度学习 API，快速构建深度学习原型的框架！对于一个高层的深度学习框架，必须照顾到用户大量的数据预处理需求。keras.preprocessing 模块主要负责在获得数据后，对缺失数据的填充，多余数据的裁剪，以及使得数据适配深度网络输入层的一些规范处理。这些常用的方法避免了初学深度学习的用户自己重头开始写预处理方法的困扰。本章将讲解 Keras 提供的常用的数据预处理工具方法，以及对于特定数据预处理使用场景的解析。

6.1　序列数据预处理

本节介绍 Keras 提供的常用序列数据预处理方法及其使用要点。

6.1.1　序列数据填充

在数据预处理阶段，经常要在序列中填充缺失值，或者裁剪一些多余的值，还可能对于每个序列加一个长度限制，这时就需要 pad_sequences 方法。

（1）pad_sequences 定义：

```
    keras.preprocessing.sequence.pad_sequences(sequences,maxl
en=None,dtype='int32', padding='pre', truncating='pre', value=
0.)
```

该方法将集合中的每个序列填充为等长。如果提供了 maxlen 参数，首先会裁剪每个序列的长度最长为 maxlen。裁剪动作可以发生在序列起始（默认）或结尾。如果没有提供 maxlen 参数，就不裁剪。最后该方法会自动填充每个序列为等长（默认在序列起始填充）。

（2）参数。

- sequences：浮点数或整数构成的两层嵌套列表（每个元素序列的列表）。
- maxlen：None 或整数，为序列的最大长度。大于此长度的序列将被截短。
- dtype：返回的 Numpy.ndarray 的数据类型。

- padding：取值为 pre 或 post，表示当需要填补序列时，是在序列的起始补还是结尾补。
- truncating：取值为 pre 或 post，表示当需要裁剪序列时，是从序列的起始断还是结尾截断。
- value：浮点数，此值将在填充时代替默认的填充值 0。

（3）返回值：返回形如 (number_of_sequences, maxlen) 的 Numpy 2D 张量。

为了展示调用方法，首先导入将要讲解的三个预处理方法。

```
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.sequence import make_sampling_
table
from keras.preprocessing.sequence import skipgrams
```

接下来举例说明 pad_sequences 的具体使用，首先测试默认填充：

```
def test_pad_sequences():
    a = [[1], [1, 2], [1, 2, 3]]

    # test padding
    b = pad_sequences(a, maxlen=3, padding='pre')
print(b, ", shape: " , b.shape)
    b = pad_sequences(a, maxlen=3, padding='post')
    print(b, ", shape: " , b.shape)
```

输出如下：

```
[[0 0 1]
 [0 1 2]
 [1 2 3]] , shape:  (3, 3)
[[1 0 0]
```

```
 [1 2 0]
 [1 2 3]] , shape:  (3, 3)
```

测试裁剪：

```
    # test truncating
    b = pad_sequences(a, maxlen=2, truncating='pre')
print(b, ", shape: " , b.shape)
    b = pad_sequences(a, maxlen=2, truncating='post')
    print(b, ", shape: " , b.shape)
```

输出如下：

```
[[0 1]
 [1 2]
 [2 3]] , shape:  (3, 2)
[[0 1]
 [1 2]
 [1 2]] , shape:  (3, 2)
```

测试填充指定的特定值：

```
    # test value
      b = pad_sequences(a, maxlen=3, value=1)
      print(b, ", shape: " , b.shape)
```

输出如下：

```
[[1 1 1]
 [1 1 2]
 [1 2 3]] , shape:  (3, 3)
```

代码中用 padding 参数控制了填补 0 值的填充位置。如果使用 pre 参数，在序列起始填充；如使用 post 参数，在序列结尾填充。如果使用 maxlen 参数，搭配 truncating 参数，可以控制裁剪作用的位置。如果使用 pre 参数，在序列起始裁剪；如果使用 post 参数，在序列结尾裁剪。代码最后

用 value=1 填充了序列而不是默认的 0。需要注意以下两点：

- 如果使用 maxlen 参数，当同时有“裁剪”和“填充”序列的情况，pad_sequences 会先进行“裁剪”操作，后进行“填充”操作。
- 所有序列操作默认都是在 numpy 数据序列上进行。

6.1.2 提取序列跳字样本

在自然语言处理领域，词向量（word2vec）因其自身蕴含深度学习属性，近年来备受学者和相关工程研究人员的推崇。词向量的一种模型跳字模型（Skip-gram）是一种根据单个单词预测相关上下文单词的模型。

跳字模型在训练阶段需要输入每个单词的上下文关系特征，这种特征通过提取和计算文本中 skipgrams 跳字样本获得。本质上，跳字样本是成对出现的单词组样本。如图 6-1 所示，其中 w(t)是一个单词输入，PPOJECTION 层代表了词向量隐层对单词的编码。w(t-2)、w(t-1)、w(t+1)，以及 w(t+2)都是 w(t)的上下文单词。我们可以通过采样文本中临近的单词对（即 skipgrams 跳字样本），训练上述模型，从而预测一个单词临近的上下文单词。例如，单词对（w(t)，w(t+1)）、（w(t-1)，w(t)）、（w(t-1)，w(t+1)）都是在上下文窗口中的单词对，即跳字样本。

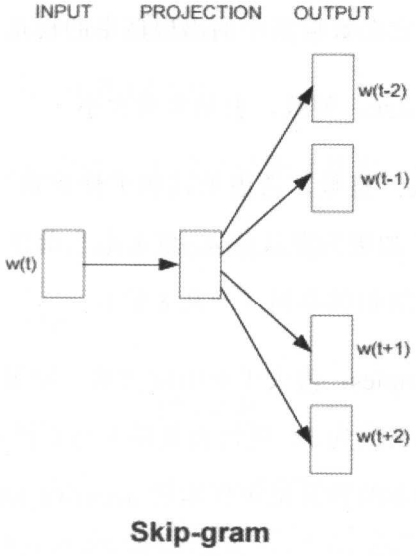


图 6-1　Skip-gram 模型原理图

在训练 Skip-gram 模型时，Keras 提供了提取跳字样本的方法 skipgrams。

（1）skipgrams 定义：

```
    keras.preprocessing.sequence.skipgrams(sequence,
vocabulary_size, window_size=4, negative_samples=1., shuffle=
True, categorical=False, sampling_table=None)
```

该方法将输入一串单词下标的序列，返回两个序列，第一个序列包含所有提取的单词关系对，另一个序列包含所有这些单词关系对是否是“上下文相关的”。如果该单词对是“上下文相关的”，则用“1”表示；否则，用“0”表示。

（2）参数。

- sequence：单词下标列表，如果使用 sampling_table，则某个词的

下标应该为它在数据集中的出现概率的权重顺序（从 1 开始）。

- vocabulary_size：整数，表示字典大小。
- window_size：整数，考虑单词相关性的窗口大小（其实是一半窗口大小）。如果为默认值 4，仅考虑当前单词向左 4 位和向右 4 位的窗口内的相邻单词（一共 8 位）。
- negative_samples：值大于 0 的浮点数。如果值为 0，则代表没有负样本；如果值为 1，则代表负样本与正样本数目相同，以此类推（即负样本的数目是正样本的 negative_samples 倍）。
- shuffle：布尔值，确定是否随机打乱样本。
- categorical：布尔值，确定是否要求返回的标签具有确定类别。
- sampling_table：长度为 vocabulary_size 的 numpy array 1D 序列，其中 sampling_table[i]代表采样到该下标为 i 的单词的概率（假定该单词是数据集中第 i 个常见的单词）。

（3）返回值：函数的输出是一个 (couples, labels) 的元组。

- couples 是一个长为 2 的整数列表：[word_index, other_word_index]，word_index 代表当前单词的下标，other_word_index 代表可能与该单词有关的相邻上下文单词下标。
- labels 是一个仅由 0 和 1 构成的列表，1 代表 other_word_index 在 word_index 的上下文窗口中，0 代表 other_word_index 是词典里的随机单词（即不在上下文窗口中）。

- 如果设置 categorical 为 True，则标签将以 one-hot 的方式给出，即 1 变为[0,1]，0 变为[1,0]。

现在使用 skipgrams 提取下面文字的跳字样本：

The fox is brown.

首先编码为如下下标：

[0,1,2,3]

接下来就可以从这句文字中提取 skipgrams：

```
def my_test_skipgrams():
    couples, labels = skipgrams([0,1,2,3], vocabulary_ size=4)
    print("couples: ", couples)
    print("labels: ", labels)
```

输出如下：

```
couples:  [[1, 2], [2, 1], [1, 3], [2, 3], [1, 1], [3, 1],
[3, 1], [1, 3], [2, 1], [3, 2], [2, 2], [3, 3]]
labels:  [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0]
```

可见默认窗口为 4，采样所有在窗口内的邻近单词对，另外，正样本数默认等于负样本数，所以，这里正样本数为 6，负样本数也为 6。（第一个单词“The”不考虑，所以 0 不出现在任何相关单词对中。）

6.1.3　生成序列抽样概率表

Keras 提供了 make_sampling_table 函数生成了一组序列单词所需的抽样概率，其中每个单词按频率表排序。即可以产生 skipgrams 函数中所需要的 sampling_table 参数。返回了一个长为 size 的向量，sampling_table[i]

代表采样到数据集中第 i 个常见的词的概率。为平衡起见，对于越是经常出现的词，要以越低的概率采到它。该函数根据 Zipf 定律计算得出。Zipf 定律认为：在自然语言的语料库里，一个单词出现的次数与它在频率表里的排名成反比。

（1）make_sampling_table 定义：

```
keras.preprocessing.sequence.make_sampling_table(size,
sampling_factor=1e-5)
```

（2）参数。

- size：表示词典的大小。
- sampling_factor：此值越低，则代表采样时更缓慢的概率衰减（即常用的词会被以更低的概率被采到）。如果设置为 1，则代表不进行下采样，即所有样本被采样到的概率都是 1。

（3）返回值：大小为 size 的序列，表示按频率表排序的单词所需的抽样概率。

最后，查看运行效果：

```
def test_make_sampling_table():
    a = make_sampling_table(5)
    print(a)
```

输出如下：

```
[  0.00315225    0.00315225    0.00547597    0.00741556
0.00912817]
```

值得注意的是，频率表排名第一的单词被考虑为与排名第二的抽样概

率相同。同时，每个单词的抽样概率递增，即与出现频率成反比。

6.2　文本预处理

在深度学习运用到自然语言处理前，文本预处理是必不可少的步骤，一些常用的工具如句子句子分割、OneHot 编码、分词器都已经集成在 Keras 预处理工具中，用户无需自己重新“造轮子”。

6.2.1　分割句子获得单词序列

text_to_word_sequence 函数将一个句子拆分成单词构成的序列。

（1）text_to_word_sequence 定义：

```
keras.preprocessing.text.text_to_word_sequence(text,filte
rs=base_filter(), lower=True, split=" ")
```

（2）参数。

- text：字符串，表示待处理的文本。
- filters：需要滤除的字符的列表或连接形成的字符串，如标点符号和空格。默认值为 base_filter()，包含标点符号、制表符和换行符等。
- lower：布尔值，是否将序列设为小写形式。
- split：字符串，单词的分隔符，如空格或分号。

（3）返回值：字符串列表。

运行测试示例代码如下：

```
    from keras.preprocessing.text import Tokenizer, one_hot ,
text_to_word_sequence
    import numpy as np

    def test_text_to_word_sequence():
        sequence=text_to_word_sequence('The cat sat on the mat. \
                                        The dog sat on the log. \
                                        Dogs and cats living together.')
        print(sequence)
```

输出如下：

```
    ['the', 'cat', 'sat', 'on', 'the', 'mat', 'the', 'dog', 'sat',
'on', 'the', 'log', 'dogs', 'and', 'cats', 'living', 'together']
```

6.2.2 OneHot 序列编码器

one_hot 函数将一段文本编码为 one-hot 形式的代码，即仅记录词在词典中的下标。

（1）one_hot 定义：

```
    keras.preprocessing.text.one_hot(text,  n,  filters=base_
filter(), lower=True, split=" ")
```

（2）参数 n：整数，表示字典长度。

（3）返回值：整数列表，每个整数是[1,n]之间的值，代表一个单词（不保证唯一性，即如果词典长度不够，不同的单词可能会被编为同一个码）。

示例代码如下：

```
    def test_one_hot():
```

```
text = 'The cat sat on the mat.'
encoded = one_hot(text, 5)
print(encoded)
```

输出如下：

```
[2, 1, 4, 4, 2, 2]
```

one_hot 函数内部是用 hash 函数生成编码，所以不保证每次结果相同。

6.2.3　单词向量化

对于文本，在深度学习实际训练中，需要将所有单词向量化（标记化），把每个单词进行编码后传送到深度网络的输入层。Keras 提供了一个 Tokenizer 类，用于向量化文本，或将文本转换为序列（即单词在字典中的下标构成的列表，从 1 算起）的类。

（1）Tokenizer 类定义：

```
keras.preprocessing.text.Tokenizer(num_words=None,filters
=base_filter(),lower=True,  split="  ",  char_level=False,
**kwargs)
```

（2）参数。

- num_words：None 或整数，处理的最大单词数量。若被设置为整数，则 Tokenizer 将被限制为处理数据集中最常见的 num_words 个单词。
- char_level：如果默认为 False，把单词作为最小单元生成 Token 向量；如果设置为 True，把字母作为最小单元生成 Token 向量。
- kwargs：处理用户误传的参数。

- 剩余与 text_to_word_sequence 同名参数含义相同。

（3）类方法。

- fit_on_texts(texts)

texts：要用以训练的文本列表。

- texts_to_sequences(texts)

texts：待转为序列的文本列表。

返回值：序列的列表，列表中每个序列对应一段输入文本。

- texts_to_sequences_generator(texts)

本函数是上述 texts_to_sequences 的生成器版本。

texts：待转为序列的文本列表。

返回值：每次调用返回对应于一段输入文本的序列。

- texts_to_matrix(texts, mode)

texts：待向量化的文本列表。

mode：“binary”“count”“tfidf”“freq”之一，默认为“binary”。

返回值：形如（len(texts), num_words）的 numpy array，即以矩阵的形式表达文本序列的特征。

- fit_on_sequences(sequences)

sequences：要用以训练的序列列表。

- sequences_to_matrix(sequences)

sequences：待向量化的序列列表。

mode：“binary”“count”“tfidf”“freq”之一，默认为“binary”。

返回值：形如（len(sequences), num_words）的 numpy array。

（4）属性。

- word_counts：字典，将单词（字符串）映射为它们在训练期间出现的次数。仅在调用 fit_on_texts 之后设置。
- word_docs：字典，将单词（字符串）映射为它们在训练期间所出现的文档或文本的数量。仅在调用 fit_on_texts 之后设置。
- word_index：字典，将单词（字符串）映射为它们的排名或者索引。仅在调用 fit_on_texts 之后设置。
- document_count：整数。分词器被训练的文档（文本或者序列）数量。仅在调用 fit_on_texts 或 fit_on_sequences 之后设置。

下面是运行测试示例代码，用于测试 fit_on_texts。

```
def test_tokenizer():
    texts = ['The cat sat on the mat.',
             'The dog sat on the log.',
             'Dogs and cats living together.']
    tokenizer = Tokenizer(num_words=10)
    tokenizer.fit_on_texts(texts)
    print('word_counts: ', tokenizer.word_counts)
    print('word_docs: ', tokenizer.word_docs)
    print('word_index: ', tokenizer.word_index)
    print('document_count: ', tokenizer.document_count)
```

输出如下：

```
    word_counts:  {'the': 4, 'on': 2, 'log': 1, 'dogs': 1,
'living': 1, 'mat': 1, 'cats': 1, 'together': 1, 'sat': 2, 'and':
1, 'dog': 1, 'cat': 1}

    word_docs:  {'the': 2, 'on': 2, 'log': 1, 'dogs': 1, 'living':
1, 'mat': 1, 'cats': 1, 'together': 1, 'sat': 2, 'and': 1, 'dog':
1, 'cat': 1}

    word_index:  {'the': 1, 'on': 2, 'log': 4, 'dogs': 5, 'living':
6, 'mat': 7, 'cats': 8, 'together': 9, 'sat': 3, 'and': 10, 'dog':
11, 'cat': 12}

    document_count:  3
```

继续测试序列生成器：

```
        sequences = []
        for  seq  in  tokenizer.texts_to_sequences_generator
(texts):
            sequences.append(seq)
        print(sequences)
```

输出如下：

```
    [[1, 3, 2, 1, 7], [1, 3, 2, 1, 4], [5, 8, 6, 9]]
```

测试文本序列以矩阵的形式表达特征，且分别用比特位 binary、count 计数、ifidf 统计，以及词频统计的模式展示。

```
        tokenizer.fit_on_sequences(sequences)

        for mode in ['binary', 'count', 'tfidf', 'freq']:
            matrix = tokenizer.texts_to_matrix(texts, mode)
            print(mode, " : ", matrix)
```

输出如下：

```
binary  :
[[ 0.  1.  1.  1.  0.  0.  1.  0.  1.  0.]
 [ 0.  1.  1.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  1.  0.  1.  0.  1.]]

count  :
[[ 0.  2.  1.  1.  0.  0.  1.  0.  1.  0.]
 [ 0.  2.  1.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  1.  0.  1.  0.  1.]]

tfidf  :
[[ 0.         1.17360019  0.69314718  0.69314718  0.          0.
   0.91629073  0.          0.91629073  0.        ]
 [ 0.         1.17360019  0.69314718  0.69314718  0.          0.
   0.          0.          0.          0.        ]
 [ 0.         0.          0.          0.          0.91629073  0.91629073
   0.          0.91629073  0.          0.91629073]]

freq  :
[[ 0.         0.33333333  0.16666667  0.16666667  0.          0.
   0.16666667  0.          0.16666667  0.        ]
 [ 0.         0.5         0.25        0.25        0.          0.
   0.          0.          0.          0.        ]
 [ 0.         0.          0.          0.          0.25        0.25
   0.          0.25        0.          0.25      ]]
```

6.3　图像预处理

Keras 图片预处理工具提供了两个很强大的功能：图片增强功能和样本无限生成功能。所有的图片预处理工具都写在一个类 ImageDataGenerator 中。该类的主要功能是不断生成一个 batch 的图像数

据，支持实时数据增强。数据增强包括均值化、白化样本图片，也包括对图片的旋转、平移、伸缩、翻转等变换。训练时该函数会无限生成数据，直到达到规定的 epoch 次数为止。

通过 Keras 图片增强功能，可以有效提高图片利用率，这在一定程度上提高了深度网络的性能，防止过拟合（同一张图片的不同变换保证了模型在更多情况下对该图片的拟合）。

（1）ImageDataGenerator 类定义：

```
    keras.preprocessing.image.ImageDataGenerator(featurewise_
center=False,
        samplewise_center=False,
        featurewise_std_normalization=False,
        samplewise_std_normalization=False,
        zca_whitening=False,
        rotation_range=0.,
        width_shift_range=0.,
        height_shift_range=0.,
        shear_range=0.,
        zoom_range=0.,
        channel_shift_range=0.,
        fill_mode='nearest',
        cval=0.,
        horizontal_flip=False,
        vertical_flip=False,
        rescale=None,
        data_format=K.image_data_format())
```

（2）参数。

- featurewise_center：布尔值，使输入数据集去中心化（均值为 0），按 feature 执行。

- samplewise_center：布尔值，使输入数据的每个样本均值为 0。
- featurewise_std_normalization 布尔值，将输入除以数据集的标准差以完成标准化，按 feature 执行。
- samplewise_std_normalization：布尔值，将输入的每个样本除以其自身的标准差。
- zca_whitening：布尔值，对输入数据施加 ZCA 白化（ZCA 即零相位成分分析）。
- rotation_range：整数，表示数据增强时图片随机转动的角度范围为（-rotation_range, rotation_range），即无论顺时针或者逆时针旋转，不超过 rotation_range 角度。
- width_shift_range：浮点数，表示图片宽度的某个比例，数据增强时图片水平偏移的幅度。
- height_shift_range：浮点数，表示图片高度的某个比例，数据增强时图片竖直偏移的幅度。
- shear_range：浮点数，表示剪切强度（逆时针方向的剪切变换角度）。
- zoom_range：浮点数或形如[lower,upper]的列表，随机缩放的幅度，若为浮点数，则相当于在[lower,upper] = [1 - zoom_range, 1+zoom_range]这样的区间缩放。
- channel_shift_range：浮点数，表示随机通道偏移的范围。

- fill_mode：取值为 constant、nearest、reflect 或 wrap，默认值为 nearest。当进行变换时超出边界的点将根据本参数给定的方法进行处理。
- cval：浮点数或整数，当 fill_mode=constant 时，指定向超出边界的点填充的值，默认填充 0。
- horizontal_flip：布尔值，进行随机水平翻转。
- vertical_flip：布尔值，进行随机竖直翻转。
- rescale：重放缩因子，默认为 None。如果为 None 或 0，则不进行放缩，否则会将该数值乘到数据上（在应用其他变换之前）。
- data_format：字符串，取值为“channel_first”或“channel_last”，代表图像的通道维的在向量中的位置。以 128×128 的 RGB 图像为例，“channel_first”对应数据向量为（3,128,128），而“channel_last”对应数据向量为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若未设置过，则默认为“channel_last”。

（3）类方法。

- fit(X, augment=False, rounds=1)：计算依赖于数据的变换所需要的统计信息（均值方差等），只有使用 featurewise_center，featurewise_std_normalization 或 zca_whitening 时需要此函数。
  - ✧ X：numpy array，样本数据，秩应为 4。在黑白图像的情况下，channel 轴的值为 1；在彩色图像情况下，channel 轴的

值为 3。

- ✧ augment：布尔值，确定是否使用随即提升过的数据。
- ✧ round：若设 augment=True，确定要在数据上进行多少轮数据增强，默认值为 1。
- ✧ seed：整数，随机数种子。

- flow(self,X, y, batch_size=32, shuffle=True, seed=None, save_to_dir=None, save_prefix='', save_format='jpeg')：接收 numpy 数组和标签为参数，生成经过数据增强或标准化后的 batch 数据，并在一个无限循环中不断地返回 batch 数据。
  - ✧ X：样本数据，秩应为 4。在黑白图像的情况下，channel 轴的值为 1；在彩色图像情况下，channel 轴的值为 3。
  - ✧ y：标签。
  - ✧ batch_size：整数，默认为 32。
  - ✧ shuffle：布尔值，是否随机打乱数据，默认为 True。
  - ✧ save_to_dir：None 或字符串，该参数能提升后的图片保存起来，用以可视化。
  - ✧ save_prefix：字符串，保存提升后图片时使用的前缀，仅当设置了 save_to_dir 时生效。
  - ✧ save_format：取值为 png 或 jpeg，用于指定保存图片的数据格式，默认为 jpeg。

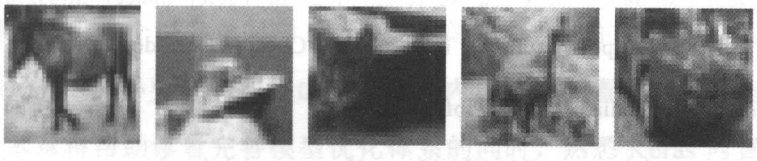
- ✧ seed：整数，表示随机数种子数。
- ✧ 返回：形如(x,y)的 tuple，x 是代表图像数据的 numpy 数组，y 是代表标签的 numpy 数组。该迭代器无限循环。

- flow_from_directory(directory)：以文件夹路径为参数，生成经过数据增强/归一化后的数据，在一个无限循环中无限产生 batch 数据。
  - ✧ directory：目标文件夹路径。对于每一个类，该文件夹都要包含一个子文件夹。子文件夹中任何 JPG、PNG 和 BNP 的图片都会被生成器使用。
  - ✧ target_size：整数 tuple，默认为（256, 256），图像将被 resize 成该尺寸。
  - ✧ color_mode：颜色模式为 grayscale 或 rgb，默认为 rgb。代表这些图片是否会被转换为单通道或三通道的图片。
  - ✧ classes：可选参数，为子文件夹的列表，如['dogs','cats']默认为 None。若未提供，则该类别列表将自动推断（类别的顺序将按照字母表顺序映射到标签值）。
  - ✧ class_mode：取值为 categorical、binary、sparse 或 None。默认为 categorical。该参数决定了返回的标签数组的形式，如果设置为 categorical，则返回 2D 的 one-hot 编码标签；如果设置为 binary，则返回 1D 的二值标签；如果设置为 sparse，则返回 1D 的整数标签；如果设置为 None，则不返回任何标

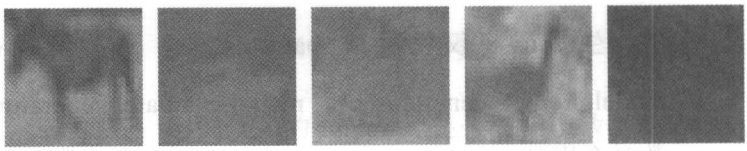
签，生成器将仅仅生成 batch 数据，这种情况在使用 model.predict_generator()和 model.evaluate_generator()等函数时会用到。

- ✧ batch_size：表示 batch 数据的大小，默认为 32。
- ✧ shuffle：表示是否打乱数据，默认为 True。
- ✧ seed：可选参数，表示打乱数据和进行变换时的随机数种子。
- ✧ save_to_dir：None 或字符串，该参数能将增强后的图片样本保存起来，用于可视化。
- ✧ save_prefix：字符串，保存经过旋转、平移、翻转等增强操作后的图片时使用的前缀，仅当设置了 save_to_dir 时生效。
- ✧ save_format：取值为 png 或 jpeg，表示指定保存图片的数据格式，默认为 jpeg。
- ✧ follow_links：表示是否访问子文件夹中的软链接。

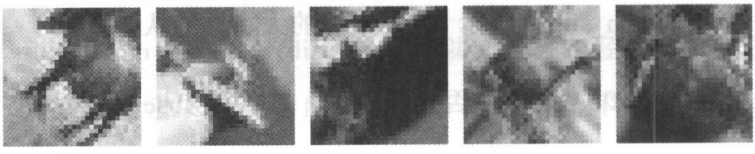
我们用 Keras 图像预处理的工具类，对 cifar-10 图片进行预处理的一些结果如图 6-2 所示。



a)　Cifar-10 原图（类别分别为：“马”，“船”，“猫”，“鸟”，“车”）

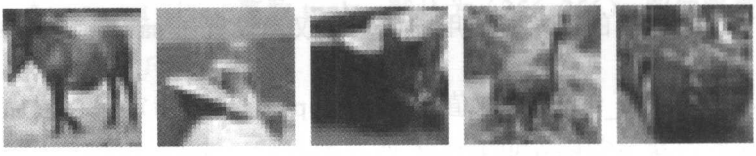


b) samplewise_center 均值化处理后



c) rotation_range 90 度角旋转处理后

d) width_shift_range 水平偏移效果



e) horizontal_flip 随机进行水平翻转效果

图 6-2 对图像进行预处理

参考文献

[1] Efficient Estimation of Word Representations in Vector Space，https://arxiv.org/pdf/1301.3781v3.pdf，Author：T Mikolov, K Chen, G Corrado, J Dean

[2] https://en.wikipedia.org/wiki/Zipf%27s_law

[3] http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

第 7 章

Keras 内置网络配置

如果把一个完整的深度学习项目比作火箭发射项目，仅仅有火箭骨架（网络各层拓扑结构）和燃料（训练数据）是远远不够的。在发射火箭，探索宇宙前，至少还需考虑以下这些要素：

（1）制定此次火箭发射的最终任务（模型性能评估 Metrices 和损失函数 Losses）。

（2）选择火箭引擎（优化器 Optimizer 和激活函数 Activation）。

（3）选择火箭发射地点（初始化参数 Initializers）。

（4）火箭巡航控制（正则项 Regularizer 和约束项 Constraint）。

因此对于一个深度学习项目，想要训练出一个好的模型，对应不同的应用和场景，势必要选择和考虑以上这些内置网络配置，以实现对模型的优化。本章将帮助读者完善模型优化体系的同时，熟悉 Keras 内置网络配置方法。

7.1 模型性能评估模块

我们之所以把模型性能评估 **Metrices** 放在首要考虑，是因为在特定应用中，实践深度学习首先应该制定一个初步的训练目标。在模型训练前期就充分考虑模型性能评估，能够有效地节省模型调优参数的时间，同时针对训练任务容易把握住出现的问题，有的放矢。

比如，对于预测要求不高的应用，可以选择 top_k_categorical_accuracy，即保证前 k 类的分类可信度即可；对于普通的平衡数据集，可能 categorical_crossentropy 即交叉熵损失函数就能满足要求；对于输出类别非常庞大，如有 50 000 个标签的稀疏预测，sparse_categorical_accuracy 即稀疏分类预测更合适；而对于严重的非平衡数据集，可能就需要更复杂的混淆矩阵来分析。

Keras 集成了很多重要的模型评估方法。这些内置的方法满足了大部分的模型评估方法。如果这些内置方法都不能满足训练目标，本节最后将介绍如何定制自己的模型评估方法。

7.1.1 Keras 内置性能评估方法

Keras 2.0.3 内置的评估方法，大致分为 3 种：一般类评估、稀疏类评估和模糊评估。

1. 一般类评估

- binary_accuracy：对二分类问题，计算在所有预测值上的平均正确率。

- categorical_accuracy：对多分类问题，计算在所有预测值上的平均正确率。
- mean_squared_error：即 MSE，均方误差。
- mean_absolute_error：即 MAE，平均绝对误差。
- mean_absolute_percentage_error：即 MAPE，平均绝对百分比误差。
- mean_squared_logarithmic_error：即 MSLE，均方对数误差。
- hinge：铰链误差。
- squared_hinge：平方铰链误差。
- binary_crossentropy：二分类交叉熵，亦称作对数损失，logloss。
- categorical_crossentropy：多分类交叉熵，亦称作多类的对数损失，注意使用该目标函数时，需要将标签转化为形如（nb_samples, nb_classes）的二值序列。
- poisson：泊松损失，（predictions - targets * log(predictions)）的均值。
- cosine_proximity：用余弦来判断两个向量的相似度。
- kullback_leibler_divergence：又称为 KL-divergence（KL-散度），信息增益，信息散度，用来度量两个分布 P 和 Q 的相似度。

2. 稀疏类评估

- sparse_categorical_accuracy：与 categorical_accuracy 相同，在对

稀疏的目标值预测时有用。

- sparse_categorical_crossentropy：与 categorical_crossentropy 相同，在对稀疏的目标值预测时有用。使用该函数时仍然需要标签与输出值的维度相同，同时需要在标签数据上增加一个维度，即 np.expand_dims(y,-1)。

3. 模糊评估

top_k_categorical_accracy：计算 top-k 正确率，当预测值的前 k 个值中存在目标类别即认为预测正确。

7.1.2 使用 Keras 内置性能评估

在第 7.1.1 节中提到的模型评估函数在模型编译时由 metrices 关键字设置。性能评估函数类似于损失函数，只不过该性能的评估结果不会用于训练，也不会影响训练，只是在训练时提供参考。具体的使用方法有以下两种。

（1）通过字符串来使用域定义的性能评估函数。

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['mae', 'acc'])
```

（2）自定义一个 Theano/TensorFlow 函数并使用。

```
from keras import metrics

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=[metrics.mae,
```

```
metrics.categorical_accuracy])
```

上述代码的终端执行训练效果如下：

```
    $ python mnist_cnn.py
    Epoch 1/12
    60000/60000 [==============================] - 841s - loss:
0.3485 - mean_absolute_error: 0.0339 - acc: 0.8950 - val_loss:
0.0831 - val_mean_absolute_error: 0.0086 - val_acc: 0.9747
```

可以注意到，Keras 进度条中多了一项 mean_absolute_error 性能评估。其中，loss（训练损失值）、acc（训练准确率）、val_loss（测试损失值），以及 val_acc（测试准确率）都是 Keras 框架默认自带的性能评估。

7.1.3　自定义性能评估函数

自定义评估函数可以在模型编译时传入。该函数应该以(y_true, y_pred)为参数，并返回单个张量，或从 metric_name 映射到 metric_value 的字典。下面是一个示例代码。

```
    import keras.backend as K

    def mean_pred(y_true, y_pred):
        return K.mean(y_pred)

    model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy', mean_pred])
```

7.2　损失函数

本质上，我们可以定义损失函数为：参与了实际模型训练的性能评估

函数。

无论是从理论层面还是代码层面，第 7.1 节中的性能评估函数和本节的损失函数是同一个函数，唯一的区别是：是否把该函数作为目标函数去训练模型？如果“是”，则该损失函数就决定了模型每一步迭代的优化方向，所以必须慎重选用损失函数。

Keras 内置损失函数的使用和内置评估函数使用一样，有以下两种方法。

（1）通过传递预定义目标函数名字指定目标函数。

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

（2）自定义一个 Theano/TensorFlow 函数并使用。

```
from keras import losses

model.compile(loss=losses.mean_squared_error,  optimizer=
'sgd')
```

注意：Keras 模型编译时，loss 是必填参数，否则模型无法构造并导致错误。

```
Traceback (most recent call last):
  File "mnist_cnn.py", line 60, in <module>
    metrics=['accuracy'])
TypeError: compile() missing 1 required positional argument:
'loss'
```

因为损失函数是性能评估函数的子集，我们仅作罗列，不加赘述，详情见 7.1.1 节。

Keras 内置支持的损失函数如下：

- mean_squared_error 或 mse。
- mean_absolute_error 或 mae。
- mean_absolute_percentage_error 或 mape。
- mean_squared_logarithmic_error 或 msle。
- squared_hinge。
- hinge。
- binary_crossentropy。
- categorical_crossentropy。
- sparse_categorical_crossentrop。
- kullback_leibler_divergence。
- poisson。
- cosine_proximity。

注意：当使用 categorical_crossentropy 作为目标函数时，标签应该为多类模式，即 one-hot 形式编码的向量，而不是单个数值。用户可以使用工具中的 to_categorical 函数完成该转换。一个示例代码如下：

```
    from keras.utils.np_utils import to_categorical
    int_labels = [1,2,3]
    categorical_labels=to_categorical(int_labels, num_classes
= None)
    print(categorical_labels)
```

输出如下：

```
[[ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

7.3 优化器函数

优化器（Optimizer）是整个深度学习网络训练的“引擎”。选定了整个深度网络的损失函数，紧接着需要考虑的就是优化器的选择。因为有了训练目标，剩下最重要的就是达成该目标的方法。本节我们不会深入优化器和最优化方法的艰深理论，而是希望读者理解不同优化器的差异和优势所在，并且了解 Keras 对于这些优化器的支持程度。必须明确的是，虽然一些新型优化器，如 Adam 总体上较其他优化器表现更好，但是依然不能否认一些优化器在特定情况下有更大优势。在使用 Keras 尝试各种优化器的过程中，我们可对各种优化器的表现有更直观的认识。

7.3.1 Keras 优化器使用

和损失函数 loss 一样，优化器 optimizer 是编译 Keras 模型必要的参数之一。

Keras 内置优化器的使用和损失函数类似，也有两种方式。

（1）可以在调用 model.compile()之前初始化一个优化器对象，然后传入该函数。

```
from keras import optimizers
```

```
    model = Sequential()
    model.add(Dense(64, init='uniform', input_shape=(10,)))
    model.add(Activation('tanh'))
    model.add(Activation('softmax'))

    sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9,
nesterov=True)
    model.compile(loss='mean_squared_error', optimizer=sgd)
```

（2）可以在调用 model.compile() 时传递一个预定义优化器名，优化器此时的参数将使用默认值。

```
    model.compile(loss='mean_squared_error', optimizer='sgd')
```

以上两种方式效果是一致的，即使用了随机梯度下降的优化器 SGD。

值得注意的是，有两个参数是所有优化器都可以使用的参数，即 clipnorm 和 clipvalue，用于对梯度进行裁剪。一个示例代码如下：

```
    from keras import optimizers

    sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
```

clipnorm 为浮点数，以上代码把所有参数的梯度 L2 正则规范限制在最大值为 1，即如果大于 1，将被裁剪。

```
    from keras import optimizers

    sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

clipvalue 为浮点数，以上代码把所有参数的梯度限制在-0.5～0.5，即如果超出绝对值 0.5 的范围，将被裁剪。

7.3.2 Keras 内置优化器

Keras 支持大多数常用的优化器，并且支持优化器丰富的参数配置。

1. SGD

（1）定义：

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- momentum：大于 0 的浮点数，动量参数。
- decay：大于 0 的浮点数，每次更新后的学习率衰减值。
- nesterov：布尔值，确定是否使用 Nesterov 动量。

目前主流框架的随机梯度下降算法都是支持 mini-batch 的 SGD，所以此处的 SGD 不是传统只支持一个样本更新的算法，可以在 Keras 模型训练时加上 batch_size 参数指定每个 batch 训练多少样本。

momentum 项和 nesterov 项都是用动量调节的方式使梯度更新更加灵活，对不同情况有针对性。但是，动量参数和学习率是人工设置的一些超参数，是有些生硬，接下来介绍 Keras 的几种自适应学习率.

2. Adagrad

（1）定义：

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-8)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

Adagrad，全称为 adaptive gradient（自适应梯度下降）的思想很简单：为了使梯度自适应地变化，给予不经常更新的参数更大的梯度，给予经常更新的参数更小的梯度。带来的一个好处是，Adagrad 算法对稀疏数据集效果较好。然而带来的两个缺点，一个是如果内置参数设置过大，则会使正则约束过于敏感，对梯度的调节太大；另一个缺点是依赖于人工设置一个全局学习率。

3. Adadelta

（1）定义：

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- rho：大于 0 的浮点数，算法内置系数，建议为 0.9。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

Adadelta 是对 Adagrad 的扩展，最初方案依然是对学习率进行自适应约束。Adagrad 会累加之前所有的梯度平方，而 Adadelta 只累加固定大小的项，并且也不直接存储这些项，仅仅是近似计算对应的平均值，衰减学习率的同时约束 Adagrad 的敏感度。

4. RMSprop

（1）定义：

```
    keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, ,
decay=0.0)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- rho：大于 0 的浮点数，算法内置系数，建议为 0.9。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

RMSprop，全称为 Root Mean Square backpropagation（均方根后向传播）是深度学习开创者之一 Geoff Hinton 教授提出的一种自适应学习率的优化器，是为了解决 Adagrad 算法过于敏感地调整梯度的问题。除学习率可调整外，建议保持优化器的其他默认参数不变。该优化器依赖于全局学习率，适合处理非平稳目标，对于递归神经网络 RNN 效果较好。

5. Adam

（1）定义：

```
    keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=1e-08, decay=0.)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- beta_1/beta_2：浮点数，算法内置系数，0<beta<1，通常很接近 1。

- epsilon：大于 0 的小浮点数，防止除 0 错误。

Adam，全称为 Adaptive Moment Estimation（自适应动量估计），是目前各方面表现较好的优化器，结合了 Adagrad 善于处理稀疏梯度和 RMSprop 善于处理非平稳目标的优点。除了训练步长 lr 外，推荐保持其他超参数不变，即推荐 β1=0.9，β2=0.999，ϵ=1e-08。

6. Adamax

（1）定义：

```
    keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999,
epsilon=1e-08)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- beta_1/beta_2：浮点数，算法内置系数，0<beta<1，通常很接近 1。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

Adamax 优化器来自于 Adam 的论文的第 7 部分，该方法是基于无穷范数的 Adam 方法的变体。

7. Nadam

（1）定义：

```
    keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999,
epsilon=1e-08, schedule_decay=0.004)
```

（2）参数。

- lr：大于 0 的浮点数，学习率。
- beta_1/beta_2：浮点数，算法内置系数，0<beta<1，通常很接近 1。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

Nadam，全称为 Nesterov Adam optimizer，类似于带有 Nesterov 动量项的 Adam。在想使用带动量的 RMSprop，或者 Adam 的情况下，可以尝试使用 Nadam 取得更好的效果。

7.4 激活函数

激活函数（Activation）的改进，对于近年来直接监督式深度网络的突破，起到了至关重要的作用。新型激活函数 ReLU 的不饱和性，使得训练像类似 VGG 这样较深的网络成为可能。

这一节我们将学会如何在 Keras 网络中快速添加激活函数，并且整理给出 Keras 支持的所有激活。

7.4.1 添加激活函数方法

Keras 支持 3 种添加激活函数的形式。

（1）通过设置单独的激活层实现，示例如下。

```
from keras.layers import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

（2）可以在网络层对象的构造时通过传递 activation 参数实现。

```
model.add(Dense(64, activation='tanh'))
```

（3）可以通过传递一个逐元素运算的 Theano 或 TensorFlow 函数来作为激活函数。

```
from keras import backend as K

def tanh(x):
    return K.tanh(x)

model.add(Dense(64, activation=tanh))
model.add(Activation(tanh)
```

以上 3 种添加方法的效果是等同的，即在全连接层之后添加一个双曲正切 tanh 激活函数。

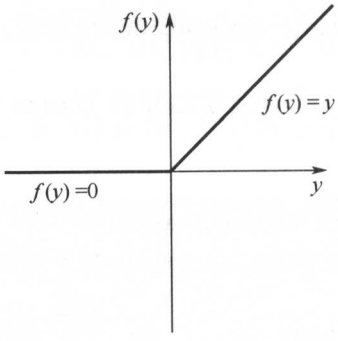
7.4.2　Keras 内置激活函数

Keras 内置的基本激活函数位于 keras.activations 模块中。

- softmax：即归一化指数函数。通常对深度网络的最后输出进行 softmax 变换，对输出进行归一化，凸显其中最大的值并抑制远低于最大值的其他分量。输入数据应形如（nb_samples, nb_timesteps, nb_dims）或（nb_samples,nb_dims）的张量，输出数据是经过 softmax 变换后的张量。

- ReLU：全称为 Rectified Linear Unit（修正线性单元），如图 7-1 所示。与传统的 sigmoid 激活函数相比，ReLU 能够有效缓解梯度消失问题，训练较深的网络，是目前深度学习最常见的激活函

数。即如果 $x<=0$，则 $f(x)=0$；如果 $x>0$，则 $f(x)=x$。



$$ReLU(x)=\begin{cases} x & if\ x>0 \\ 0 & if\ x\leqslant 0 \end{cases}$$

图 7-1　ReLU 示意图

- ELU：全称为 Exponential Linera Unit（指数线性单元），融合了 sigmoid 和 ReLU，具有左侧软饱和性（左侧有极值逼近），如图 7-2 所示。如果希望网络具有更快的收敛速度，可以考虑此优化器。此处注意 ELU 有 alpha 控制负因子的参数可以调节左侧的软饱和。

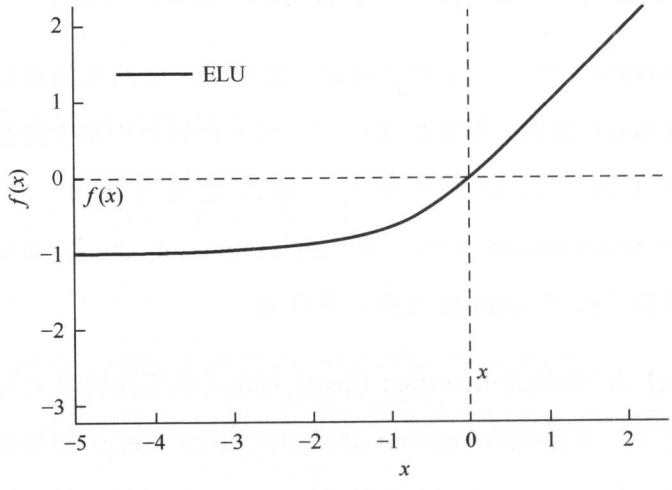


图 7-2　ELU 示意图

- softplus：softplus 是对 ReLU 的平滑逼近的解析函数形式，所以 softplus 和 ReLU 的表现非常相似。图 7-3 为 softplus 与 ReLU 比较。

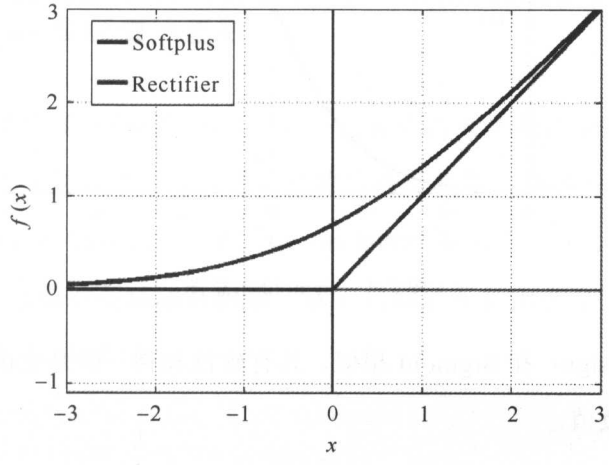


图 7-3　softplus 与 ReLU 函数比较（平滑曲线为 softplus，线性折线为 ReLU）

但是，softplus 的非线性函数使得深度网络计算偏导比较困难，而 ReLU 是完全线性的，计算偏导的效率大大提高，这也正是 Relu 比 softplus 更广泛流行的原因。

- Sigmoid：Sigmoid 是在 ReLU 出现之前使用范围最广的一类激活函数，如图 7-4 所示。Sigmoid 的饱和性虽然会导致梯度消失，不利于深度学习训练，但其也有有利的一面。例如它在物理意义上最为接近生物神经元。（0，1）的输出范围还可以被表示概率，或用于归一化。

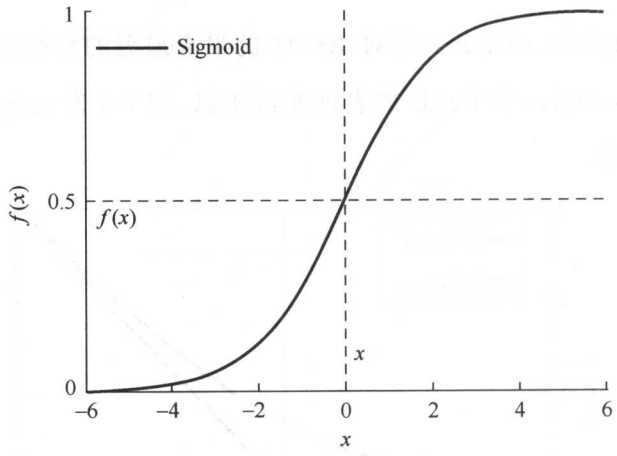


图 7-4　sigmoid 示意图

- softsign：和 Sigmoid 相似，具有软饱和性，但是它的左侧饱和值是负值。
- tanh：和 Sigmoid 相似，具有软饱和性，tanh 网络的收敛速度要比 Sigmoid 快，如图 7-5 所示。

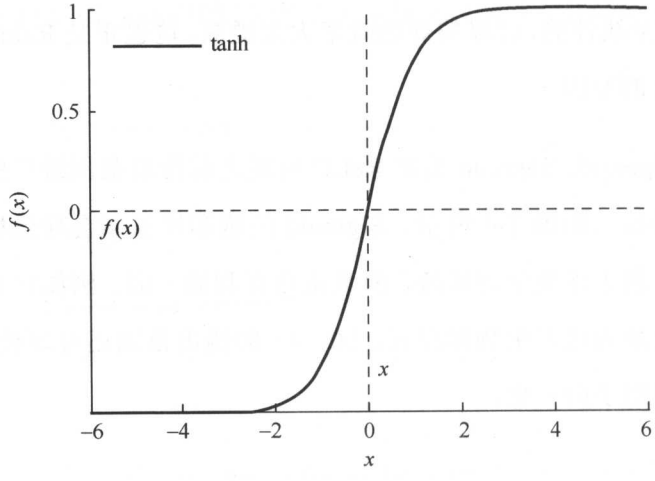


图 7-5　tanh 示意图

- hard_sigmoid：hard_sigmoid 是类似 sigmoid 的线性拟合，可提高

计算效率，但是同样不适合训练深度学习网络。

- linear：简单的线性函数，两端都是不饱和的。

7.4.3　Keras 高级激活函数

对于简单的 Theano/TensorFlow 不能表达的复杂激活函数，可通过高级激活函数来实现，如含有可学习参数的激活函数，PReLU、LeakyReLU 等。这些 Keras 高级激活函数位于 keras.layers 模块下，因此，需要用添加 layer 层的方式添加高级激活函数。以下代码展示如何在全连接层后加入高级激活函数 LeakyRelu。

```
model.add(Dense(512, 512, activation='linear'))
model.add(LeakyReLU(alpha=.001))
```

Keras 2.0.3 目前支持以下 4 种高级激活函数。

1. LeakyReLU 层

LeakyRelU 是 ReLU 的特殊版本，当不激活时，LeakyReLU 仍然会有非零输出值，从而获得一个较小梯度，避免 ReLU 可能出现的神经元“死亡”现象，如图 7-6 所示。即如果 $x<0$，则 $f(x)$=alpha * x；如果 $x>=0$，则 $f(x) = x$。其中，alpha 是一个人工指定的系数。

（1）定义：

```
keras.layers.advanced_activations.LeakyReLU(alpha=0.3)
```

（2）参数。

- alpha：大于 0 的浮点数，代表激活函数图像中第三象限线段的

斜率。

（3）输入 shape：任意，当使用该层为模型首层时需指定 input_shape 参数。

（4）输出 shape：与输入相同。

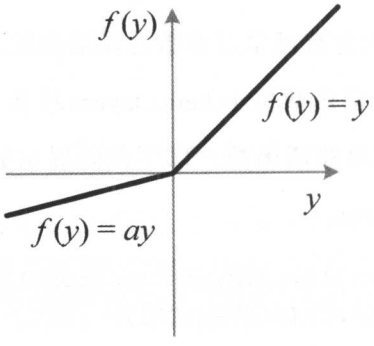


图 7-6　LeakyReLU 示意图

2. PReLU 层

该层为参数化的 ReLU（Parametric ReLU），表达式是：如果 $x < 0$，则 $f(x)$ = alpha * x；如果 $x>=0$，则 $f(x) = x$，如图 7-7 所示。注意，此处的 alpha 不同于 LeakyReLU 中人工设定的 alpha，此处的 alpha 是一个可学习的参数，一个与 x 的 shape 相同的向量。因此，虽然 LeakyReLU 和 PreLU 的激活函数图形非常相似，但是前者是不可动态训练的，后者是可动态训练的。

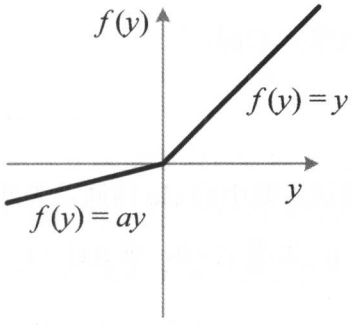


图 7-7　PRelU 示意图

（1）定义：

```
    keras.layers.advanced_activations.PReLU(alpha_initializer
='zeros',    alpha_regularizer=None,    alpha_constraint=None,
shared_axes=None)
```

（2）参数。

- alpha_initializer：alpha 的初始化函数，默认为全零初始化。
- alpha_regularizer：alpha 的正则项，默认为空。
- alpha_constraint：alpha 的约束项，默认为空。
- shared_axes：该参数指定的轴将共享同一组可学习参数，例如，假如输入特征图是从 2D 卷积过来的，具有形如（batch, height, width, channels）这样的 shape，则或许在空间域共享参数，这样每个 filter 就只有一组参数，设定 shared_axes=[1,2]可完成该目标。shared_axes 默认值为空。

（3）输入 shape：任意，当使用该层为模型首层时需指定 input_shape 参数。

（4）输出 shape：与输入相同。

3. ELU 层

此 ELU 层和内置激活函数中的 elu 相同，表达式是，如果 $x < 0$，则 $f(x)$ = alpha * (exp(x) - 1.)；如果 x>=0，则 $f(x) = x$。

（1）定义：

```
keras.layers.advanced_activations.ELU(alpha=1.0)
```

（2）参数 alpha：控制负因子的参数。

（3）输入 shape：任意，当使用该层为模型首层时需指定 input_shape 参数。

（4）输出 shape：与输入相同。

4. ThresholdedReLU 层

该层是带有阈值的 ReLU，表达式是，如果 for x > theta，则 $f(x) = x$，否则 $f(x) = 0$。

（1）定义：

```
keras.layers.advanced_activations.ThresholdedReLU(theta=
1.0)
```

（2）参数 theata：大或等于 0 的浮点数，激活阈值位置。

（3）输入 shape：任意，当使用该层为模型首层时需指定 input_shape 参数。

（4）输出 shape：与输入相同。

7.5　初始化参数

许多初学者可能会忽视在深度学习训练前初始化参数的重要性。事实上，使用一个好的初始化参数的方法，可以有效提高深度网络的整体性能。多数深度学习问题本质是一个非凸优化问题，需要找到一个相对较好的局部最优解。太小的参数容易导致权值消失（vanishing），使得模型的迭代过于缓慢；而太大的权值又容易导致梯度爆炸（exploding）或者模型很快收敛到一个相对很差的局部最优解。

目前在深度学习中，初始化参数方法提高性能已经成为一个公认的训练技巧（Trick）。基于各种初始化方法的改进和变形层出不穷，甚至在 2016 国际学习表征会议（ICLR）上发表的 LSUV 初始化方法在 cifar-10 和 cifar-100 上分别达到了 94.16%和 72.34%的精度，刷新了新的标杆记录。因此，如果期望深度网络有一个额外的性能提高，可以考虑在训练前改进初始化参数的方法。正如选择一个好的发射地，是火箭最终发射成功的重要因素之一。

7.5.1　使用初始化方法

如果不进行指定，Keras 默认对所有卷积层和全连接层进行初始化，其中，使用 glorot_uniform 作为权重初始化，并且使用全零初始化作为偏置 bias 的初始化。

如果需要指定初始化方法，不同的层可能使用不同的关键字来传递初始化方法，一般来说，指定初始化方法的关键字是 kernel_initializer 和 bias_initializer，示例如下：

```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

指定初始化方法有两种形式。

（1）初始化器可以由字符串指定（必须是 Keras 内置初始化方法之一）：

```
model.add(Dense(64, kernel_initializer='random_normal'))
```

（2）可以在层中指定一个 callable 的函数，例如：

```
from keras import initializers

model.add(Dense(64,
kernel_initializer=initializers.random_normal(stddev=0.01)))
```

以上两种方法效果是一样的，指定使用 random_normal 方法为当前全连接层权重的初始化方法。

7.5.2 Keras 内置初始化方法

Keras 目前的初始化方法都不依赖输入数据，而主要集中在针对网络结构，对参数进行合理的随机初始化编排，即不考虑训练数据对初始化参数的影响。Keras 支持的内置初始化方法如下。

1. Zeros

定义：

```
keras.initializers.Zeros()
```

全 0 初始化。最简单的初始化策略。构造全 0 矩阵作为初始权值，全零初始化在深度网络实验证明效果极差，所以谨慎使用。

2. Ones

定义：

```
keras.initializers.Ones()
```

全 1 初始化。与全 0 初始化类似。构造全 1 矩阵作为初始权值。

3. Constant

定义：

```
keras.initializers.Constant(value=0)
```

固定值初始化。与全 0 初始化类似。初始化为固定值 value。

4. RandomNormal

该方法用于正态分布初始化。

（1）定义：

```
keras.initializers.RandomNormal(mean=0.0,      stddev=0.05,
seed=None))
```

（2）参数。

- mean：均值。
- stddev：标准差。
- seed：随机数种子。

5. RandomUniform

该方法用于均匀随机分布初始化。

（1）定义：

```
keras.initializers.RandomUniform(minval=-0.05, maxval=0.05,
seed=None)
```

（2）参数：

- minval：用于均匀分布下边界。
- maxval：用于均匀分布上边界。
- seed：随机数种子。

6. TruncatedNormal

截尾高斯分布初始化，该初始化方法与 RandomNormal 类似，但位于离开均值两个标准差以外的数据将被丢弃并重新生成，形成截尾分布。该分布是神经网络权重和滤波器的推荐初始化方法。

（1）定义：

```
keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05,
seed=None)
```

（2）参数。

- mean：均值。
- stddev：标准差。
- seed：随机数种子。

7. VarianceScaling

方差缩放初始化，这个方法考虑了能够自适应目标张量的大小。当

distribution=“normal”时，样本从均值为 0，标准差为 sqrt(scale / n)的截尾正态分布中产生；当 distribution=“uniform”时，权重从[-limit, limit]范围内均匀采样，其中，limit = sqrt(3 * scale / *n*)。*n* 的取值取决于 mode 参数。

- 当 mode =“fan_in”时，*n* =权重张量的输入单元数。
- 当 mode =“fan_out”时，*n* =权重张量的输出单元数。
- 当 mode =“fan_avg”时，*n* =权重张量的输入输出单元数的均值。

（1）定义：

```
keras.initializers.VarianceScaling(scale=1.0,mode='fan_in
',distribution='normal', seed=None)
```

（2）参数。

- scale：放缩因子，正浮点数。
- mode：字符串，“fan_in”“fan_out”或“fan_avg”之一。
- distribution：字符串，“normal”或“uniform”。
- seed：随机数种子。

8. Orthogonal

该方法用于随机正交矩阵初始化。

（1）定义：

```
keras.initializers.Orthogonal(gain=1.0, seed=None)
```

（2）参数。

- gain：正交矩阵的乘性系数。
- seed：随机数种子。

9. Identity

使用单位矩阵初始化，仅适用于 2D 矩阵。

（1）定义：

```
keras.initializers.Identity(gain=1.0)
```

（2）参数

gain：正交矩阵的乘性系数。

10. lecun_uniform

LeCun 均匀分布初始化方法，参数由[-limit, limit]的区间中均匀采样获得，其中 limit=sqrt(3 / fan_in), fin_in 是权重向量的输入单元数（扇入数）。

（1）定义：

```
keras.initializers.lecun_uniform(seed=None)
```

（2）参数 seed：随机数种子。

11. glorot_normal

Glorot 正态分布初始化方法，也称作 Xavier 正态分布初始化。参数由均值为 0，标准差为 sqrt(2 / (fan_in + fan_out))的正态分布产生，其中，fan_in 和 fan_out 是权重张量的扇入、扇出（即输入和输出单元数目）。

（1）定义：

```
keras.initializers.glorot_normal(seed=None)
```

（2）参数 seed：随机数种子。

12. glorot_uniform

Glorot 均匀分布初始化方法，也称作 Xavier 均匀分布初始化，参数由[-limit, limit]的区间中均匀采样获得，其中 limit= sqrt(6 / (fan_in + fan_out), fan_in 是权重向量的输入单元数（扇入数），fan_out 是权重向量的输出单元数（扇出数）。

（1）定义：

```
keras.initializers.glorot_normal(seed=None)
```

（2）参数 seed：随机数种子。

13. he_normal

He 正态分布初始化方法，参数由 0 均值，标准差为 sqrt(2 / fan_in) 的正态分布产生，其中 fan_in 是权重向量的输入单元数（扇入数）。

（1）定义：

```
keras.initializers.he_normal(seed=None)
```

（2）参数 seed：随机数种子。

14. he_uniform

He均匀分布初始化方法，参数由[-limit, limit]的区间中均匀采样获得，其中，limit= sqrt(6 / fan_in)，fan_in 是权重向量的输入单元数（扇入数）。

（1）定义：

```
keras.initializers.he_uniform(seed=None)
```

（2）参数 seed：随机数种子。

7.5.3 自定义 Keras 初始化方法

Keras 允许自定义网络层的初始化方法。如果需要传递自定义的初始化方法，则该初始化方法必须是 callable 的，并且接收 shape（将被初始化的张量 shape）和 dtype（数据类型）两个参数，并返回符合 shape 和 dtype 的张量。示例代码如下：

```
from keras import backend as K

def my_init(shape, dtype=None):
    return K.random_normal(shape, dtype=dtype)

model.add(Dense(64, init=my_init))
```

这个自定义初始化方法的效果和内置方法 RandomNormal 的效果是相同的，都是以随机正太分布初始化。

7.6 正则项

在机器学习理论中，正则项（Regularizer）即罚函数，是对模型向量进行“惩罚”，是抵制过拟合的一种手段。正则项在实际优化过程中，对层的参数或层的激活值添加惩罚项，这些惩罚项将与损失函数一起作为网络的最终优化目标。

7.6.1　使用正则项

Keras 的正则项基于层进行惩罚，目前正则项的接口与层有关，但 Dense、Conv1D、Conv2D 和 Conv3D 具有共同的接口。

这些层有 3 个关键字参数可以施加正则项。

- kernel_regularizer：施加在权重上的正则项，继承自 keras.regularizer.Regularizer 的对象。
- bias_regularizer：施加在偏置向量上的正则项，继承自 keras.regularizer.Regularizer 对象。
- activity_regularizer：施加在输出上的正则项，继承自 keras.regularizer.Regularizer 对象。

注意：Keras 对于一些层不支持正则项约束，比如池化（pooling）层；而有一些层则有不同的正则项选项，比如递归神经网络的递归层 keras.layers.recurrent.SimpleRNN 还支持递归正则项 recurrent_regularizer，另外，batch 归一化层 keras.layers.recurrent.nomalization.BatchNormalization 支持为参数 beta 和 gamma 施加正则项 beta_regularizer 和 gamma_regularizer。

正则项使用示例代码如下：

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

以上代码在全连接层的权重和偏置上分别施加 L2 范数正则和 L1 范

数正则。

7.6.2 Keras 内置正则项

- keras.regularizers.l1(0.)：L1 范数正则化。输入参数为浮点数，L1 范数正则化因子。
- keras.regularizers.l2(0.)：L2 范数正则化。输入参数为浮点数，L2 范数正则化因子。
- keras.regularizers.l1_l2(0., 0.)：同时使用 L1 和 L2 范数正则化。输入参数为浮点数，第一个参数为 L1 范数正则化因子，第二个参数为 L2 范数正则化因子。

7.6.3 自定义 Keras 正则项

在 Keras 中可以开发自定义的正则项函数，任何以权重矩阵作为输入并返回单个数值的函数均可以作为正则项。以下是一个示例代码：

```
from keras import backend as K

def l1_reg(weight_matrix):
    return 0.01 * K.sum(K.abs(weight_matrix))

model.add(Dense(64, input_dim=64,
                kernel_regularizer=l1_reg)
```

以上代码在全连接层中使用了自定义的 L1 范数正则化项。

7.7　参数约束项

正则化抵制过拟合的方法，是在损失函数后加上一个正则项。与正则化不同的是，约束项（Constraint）是在参数输出时直接“修剪”参数，从而达到规范化和抵制过拟合的效果。

7.7.1　使用参数约束项

来自 keras.constraints 模块的函数在优化过程中为网络的参数施加约束。

约束项基于层进行约束，超出约束的参数将会被“修剪”，目前惩罚项的接口与层有关，但 Dense、Conv1D、Conv2D 和 Conv3D 具有共同的接口。

这些层通过以下关键字施加约束项。

- kernel_constraint：对主权重矩阵进行约束。
- bias_constraint：对偏置向量进行约束。

注意：Keras 对于一些层不支持约束项配置，比如池化（pooling）层；而有一些层则有不同的正则项选项，比如递归神经网络的递归层 keras.layers.recurrent.SimpleRNN 还支持递归约束项 recurrent_constraint，另外，batch 归一化层 keras.layers.recurrent.nomalization.BatchNormalization 支持为参数 beta 和 gamma 施加约束项 beta_constraint 和 gamma_constraint。

约束项使用示例代码如下：

```
from keras.constraints import maxnorm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

以上代码在全连接层的权重上施加最大模约束。

7.2.2 Keras 内置参数约束项

目前 Keras 支持 4 种内置参数约束项。

- max_norm(m=2)：最大模约束。
- non_neg()：非负性约束。
- unit_norm()：单位范数约束，强制矩阵沿最后一个轴拥有单位范数。
- min_max_norm()：最大/最小值约束，强制在一个区间范围内的约束。

参考文献

[1] I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. Book in preparation for MIT Press, 2016

[2] https://zh.wikipedia.org/wiki/%E7%9B%B8%E5%AF%B9%E7%86%B5

[3] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

[4] http://blog.csdn.net/reallocing1/article/details/56292877

[5] http://sebastianruder.com/optimizing-gradient-descent/index.html

[6] https://zhuanlan.zhihu.com/p/22252270

[7] https://arxiv.org/abs/1212.5701

[8] https://arxiv.org/abs/1412.6980v8

[9] http://cs229.stanford.edu/proj2015/054_report.pdf

[10] http://www.cs.toronto.edu/~fritz/absps/momentum.pdf

[11] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[12] https://zh.wikipedia.org/wiki/Softmax%E5%87%BD%E6%95%B0

[13] https://web.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf

[14] https://arxiv.org/pdf/1502.01852v1.pdf

[15] https://arxiv.org/pdf/1511.07289v1.pdf

[16] http://www.cnblogs.com/neopenx/p/4453161.html

[17] https://arxiv.org/abs/1312.6120

第 8 章

Keras 实用技巧和可视化

Keras 是搭建在 TensorFlow 和 Theano 之上的高层框架，因此，如果不了解 Keras 与下层交互的接口，很难进行有效的代码追踪、异常处理和超参调试。幸运的是，Keras 提供了简单易用的高层接口，方便用户进行高效的调试、排错与可视化。本章帮助读者完善 Keras 实用技巧，打通 Keras 与下层框架（Theano，Tensorflow）的使用阻隔。事实上，Keras 可以简单快捷地使用下层框架的一些调试功能，从而在没有下层知识的前提下，顺畅地实现模型原型。

8.1 Keras 调试与排错

8.1.1 Keras Callback 回调函数与调试技巧

在实际问题中，深度模型下的调试与排错非常复杂，但是，最通用和常用的方法无疑是在特定时刻设置断点、检查点，或检查日志。Keras 内置强大实用的 Callback 回调函数，满足日常的调试与排错需要。

事实上，Keras 训练实时进度条正是使用了内置的 Callback 回调函数

在每个 batch 训练完毕时给出瞬时统计信息：

```
    Epoch 1/12
    60000/60000 [==============================] - 112s - loss:
0.3129 - acc: 0.9049 - val_loss: 0.0734 - val_acc: 0.9767
    Epoch 2/12
    34048/60000 [===============>..............] - ETA: 53s -
loss: 0.1159 - acc: 0.9645
```

Keras 回调函数允许指定在训练的特定阶段调用的函数集，并使用回调函数来观察训练过程中网络内部的状态和统计信息。通过传递回调函数列表到模型的.fit()中，即可在给定的训练阶段调用该函数集中的函数。

值得注意的是，虽然称之为回调“函数”，但事实上 Keras 的回调函数是一个类，回调函数只是习惯性称呼。

接下来，我们就来学习这些回调类及其实用方法。

1. Callback 类

这是回调函数的抽象类，定义新的回调函数必须继承该类。

（1）定义：

```
keras.callbacks.Callback()
```

（2）类属性。

- params：字典，训练参数集（如信息显示 verbosity、batch 大小、epoch 数）。
- model：keras.models.Model 对象，是正在训练的模型的引用。

（3）类方法。

- on_epoch_begin：在每个 epoch 开始时调用。
- on_epoch_end：在每个 epoch 结束时调用。
- on_batch_begin：在每个 batch 开始时调用。
- on_batch_end：在每个 batch 结束时调用。
- on_train_begin：在训练开始时调用。
- on_train_end：在训练结束时调用。

注意，回调执行的先后顺序如下：

on_train_begin-> on_epoch_begin->on_batch_begin->

on_batch_end-> on_epoch_end->on_train_end

回调函数以字典 logs 为参数，该字典包含了一系列与当前 batch 或 epoch 相关的信息。

目前，模型的.fit()中有下列参数会被记录到 logs 中。

- 在每个 epoch 的结尾处（on_epoch_end），logs 将包含训练的正确率和误差，acc 和 loss，如果指定了验证集，还会包含验证集正确率和误差 val_acc 和 val_loss。注意，如果需要显示准确率 acc 和 val_acc，需要额外在.compile 中启用 metrics=['accuracy']。
- 在每个 batch 的开始处（on_batch_begin）：logs 包含 size，即当前 batch 的样本数。
- 在每个 batch 的结尾处（on_batch_end）：logs 包含 loss，若启用

accuracy，则还包含 acc。

例如，可以自定义一个回调函数，在模型训练开始时，打印模型的汇总信息。

```
    class Mylogger(keras.callbacks.Callback):
        def on_train_begin(self, logs=None):
            print('on_train_begin')

print(keras.utils.layer_utils.print_summary(self.model))

    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              verbose=1,
              validation_data=(x_test, y_test),
              callbacks=[Mylogger()])
```

上述代码定义了 Mylogger 回调函数，在开始训练时（on_train_begin）打印当前模型的汇总信息，终端输出如下：

```
    on_train_begin
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
    _________________________________________________________________
    conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
    _________________________________________________________________
    max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
    _________________________________________________________________
    dropout_1 (Dropout)          (None, 12, 12, 64)        0
    _________________________________________________________________
    flatten_1 (Flatten)          (None, 9216)              0
    _________________________________________________________________
    dense_1 (Dense)              (None, 128)               1179776
```

```
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

可见，layer_utils 模块中的 print_summary 方法给出了模型许多实用信息，包括网络各层的输出尺寸、输出通道数，以及待训练参数的数量。

2. BaseLogger 类

该回调函数用来对每个 epoch 累加性能评估（metrics）并计算其平均值。值得注意的是，该回调函数在每个 Keras 模型中都会被自动调用。

定义：

```
keras.callbacks.BaseLogger()
```

3. ProgbarLogger 类

该回调函数用来将 metrics 指定的监视指标输出到标准输出上，它也是在 Keras 模型引擎中自动调用的。

定义：

```
keras.callbacks.ProgbarLogger()
```

4. History 类

与 BaseLogger 和 ProgbarLogger 一样，该回调函数在 Keras 模型引擎中会被自动调用。history 对象实际上正是 fit 方法的返回值，即一些汇总

信息。

定义：

```
keras.callbacks.History()
```

5. ModelCheckpoint 类

（1）定义：

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_
loss', verbose=0, save_best_only=False, save_weights_only=False,
mode='auto', period=1)
```

（2）参数。

- filename：必填项，字符串，保存模型的路径。
- monitor：需要监视的值，默认为测试集损失 val_loss。
- verbose：信息展示模式，0 或 1。
- save_best_only：当设置为 True 时，将只保存在测试集上性能最好的模型。
- mode：取值为 auto、min 或 max。在 save_best_only=True 时，决定性能最佳模型的评判准则，例如，当监测值为 val_acc 时，模式应为 max；当检测值为 val_loss 时，模式应为 min。在 auto 模式下，评价准则由被监测值的名字自动推断。
- save_weights_only：若设置为 True，则只保存模型权重参数，否则将保存整个模型（包括模型结构、权重、优化器参数和其他配置信息等）。

- period：CheckPoint 之间的间隔的 epoch 数。

检查点回调函数将在每个 epoch 完成后保存模型到 filepath。示例如下：

```
    from keras.callbacks import ModelCheckpoint

    model.fit(...,

callbacks=[ModelCheckpoint(filename='./modelsave.hdf5')])
```

filepath 可以是格式化的字符串，里面的占位符将会被 epoch 值和传入 on_epoch_end 的 logs 关键字所填入。例如，filepath 若为 weights.{epoch:02d}-{val_loss:.2f}.hdf5，则会生成对应 epoch 和验证集 loss 的多个文件。示例代码如下：

```
    model.fit(...,
    callbacks=[ModelCheckpoint(filename='weights.{epoch:02d}-
{val_loss:.2f}.hdf5')])
```

保存的文件的默认格式为科学数据常用的文件格式：hdf5，方便在其他平台的迁移和还原。

6. EarlyStopping 类

提前终止训练的回调函数，当监测值不再改善时，该回调函数将中止训练。

（1）定义：

```
    keras.callbacks.EarlyStopping(monitor='val_loss',
patience=0, verbose=0, mode='auto')
```

（2）参数。

- monitor：需要监视的量，默认为 val_loss。
- patience：当提前终止被激活（如发现 loss 相比上一个 epoch 训练没有下降），则经过 patience 个 epoch 后停止训练。默认值为 0。
- verbose：信息展示模式，默认值为 0。
- mode：取值为 auto、min 或 max。在 min 模式下，如果检测值停止下降，则中止训练；在 max 模式下，当检测值不再上升，则停止训练；在 auto 模式下，方向将由函数自动根据监测变量的名称判别。默认为 auto 模式。

7. RemoteMonitor 类

该回调函数用于向服务器发送事件流，此功能需要安装 requests 库。

（1）定义：

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000')
```

（2）参数 root：该参数为根 url，默认为 root='http://localhost:9000'。回调函数将在每个 epoch 结束把产生的事件流发送到该地址，事件将被发往 path = root + '/publish/epoch/end/'。发送方法为 HTTP POST，其 data 字段的数据是按 JSON 格式编码的事件字典。

如果需要一个独立的自建服务监听事件流而不是 tensorboard 或者其他插件，这个接口将十分有用。

8. LearningRateScheduler 类

该回调函数是学习率调度器。如果写一个自定义的学习率调度器，这个类是一个不错的开始。

（1）定义：

```
keras.callbacks.LearningRateScheduler(schedule)
```

（2）参数 schedule：必填项，函数。该函数以 epoch 号为参数（从 0 算起的整数），返回一个新学习率（浮点数）。

9. TensorBoard 类

该回调函数是一个 Tensorboard 的可视化工具。

（1）定义：

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_
freq=0, write_graph=True, write_images=False, embeddings_freq=0,
embeddings_layer_names=None, embeddings_metadata=None)
```

（2）参数。

- log_dir：保存日志文件的地址，该文件将被 TensorBoard 解析以用于可视化，默认值为“./log”。
- histogram_freq：计算各个层激活值直方图的频率（每多少个 epoch 计算一次），如选择默认值为 0，则不计算。
- write_graph：是否在 Tensorboard 上可视化网络图。如果选择默认值 True，保存的目录内容可能更大。

- write_images：是否在 Tensorboard 可视化模型权重，默认选择 False。
- embeddings_freq：在每个 epoch 中被选择的嵌入层保存的频率，默认为 0，则不保存。
- embeddings_layer_names：需要被监视的嵌入层列表，如果选择默认值 None，则所有嵌入层都会被监视。
- embeddings_metadata：一个存储嵌入层名到文件名映射关系的字典，用来存储该嵌入层的元数据（metadata）。

该回调函数将日志信息写入 TensorBorad，可以动态地观察训练和测试指标的图像，以及不同层的激活值直方图。注意，使用前必须确认已安装 Tensorboard。

示例如下：

```
    tbCallBack = keras.callbacks.TensorBoard(log_dir='./Graph',
histogram_freq=0, write_graph=True, write_images=True)

    model.fit_generator(...,
                        callbacks=[tbCallBack])
```

上述示例代码首先定义了一个 TensorBoard 类的回调函数 tbCallBack，指定写入文件夹/Graph。在模型训练时，指定 callbacks=[tbCallBack]即可完成训练时的回调。

在训练时，只需把 Tensorboard 打开到指定目录即可完成实时监控训练：

```
    tensorboard --logdir=Graph --host 0.0.0.0 --port 6006
```

10. ReduceLROnPlateau 类

当评价指标不在提升时，减少学习率。

（1）定义：

```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
factor=0.1, patience=10, verbose=0, mode='auto', epsilon=0.0001,
cooldown=0, min_lr=0)
```

（2）参数。

- monitor：被监测的值，默认为 val_loss。
- factor：每次减少学习率的因子，学习率将以 lr = lr*factor 的形式被减少，factor 默认值为 0.1。
- patience：当 patience 个 epoch 过去而模型性能不提升时，学习率减少的动作会被触发，默认 patience=10 即等待 10 个 epoch 后触发动作。
- mode：取值为 auto、min 或 max，默认为 auto 模式。在 min 模式下，如果检测值停止下降，则中止训练；在 max 模式下，当检测值不再上升，则停止训练；在 auto 模式下，方向将由函数自动根据监测变量的名称判别。
- epsilon：阈值，用来确定是否进入检测值的“平原区”，默认值为 0.0001。
- cooldown：学习率减少后，会经过 cooldown 个 epoch 才重新进行正常操作，默认为 0，即马上进行正常操作。

- min_lr：学习率的下限，默认值为 0。

当学习停滞时，减少 2 倍或 10 倍的学习率常常能获得较好的效果。该回调函数用于检测指标的情况，如果在 patience 个 epoch 中看不到模型性能提升，则减少学习率。

11. CSVLogger 类

将每次 epoch 的训练结果保存在 csv 文件中，支持所有可被转换为 string 的值，包括 1D 的可迭代数值，如 np.ndarray。

（1）定义：

```
    keras.callbacks.CSVLogger(filename,          separator=',',
append=False)
```

（2）参数。

- fiename：必填项，保存的 csv 文件名，如 run/log.csv。
- separator：字符串，csv 分隔符。
- append：默认为 False，为 True 时，csv 文件如果存在，则继续写入；为 False 时，总是覆盖 csv 文件。

12. LambdaCallback 类

该工具类用于创建简单的 callback 的 callback 类。该 callback 的匿名函数将会在适当的时候被调用。

（1）定义：

```
    keras.callbacks.LambdaCallback(on_epoch_begin=None,
```

```
on_epoch_end=None,   on_batch_begin=None,   on_batch_end=None,
on_train_begin=None, on_train_end=None)
```

（2）参数。

- on_epoch_begin：在每个 epoch 开始时调用。
- on_epoch_end：在每个 epoch 结束时调用。
- on_batch_begin：在每个 batch 开始时调用。
- on_batch_end：在每个 batch 结束时调用。
- on_train_begin：在训练开始时调用。
- on_train_end：在训练结束时调用。

注意，该回调函数假定了一些位置参数。

- on_epoch_begin 和 on_epoch_end 假定输入参数是：epoch 和 logs。
- on_batch_begin 和 on_batch_end 假定输入参数是：batch 和 logs。
- on_train_begin 和 on_train_end 假定输入参数是：logs。

以下是示例代码：

```
    batch_print_callback = LambdaCallback(
        on_batch_begin=lambda batch,logs: print(batch))

    import numpy as np
    import matplotlib.pyplot as plt
    plot_loss_callback = LambdaCallback(
        on_epoch_end=lambda epoch,
logs: plt.plot(np.arange(epoch),
                       logs['loss']))
```

```
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
    p.terminate() for p in processes if p.is_alive()])

model.fit(...,
      callbacks=[batch_print_callback,
         plot_loss_callback,
         cleanup_callback])
```

8.1.2　备份和还原 Keras 模型

备份和还原模型是机器学习实践中非常实用的技巧之一。Keras 框架下提供接口方便用户备份模型拓扑和已训练的参数，不建议用户自行使用 pickle 或者 cPickle 库存储 Keras 模型。

建议用户使用 model.save(filepath) 保存 Keras 模型到一个 HDF5 格式的文件（filepath 为文件路径），可用来在其他平台如 matlab、deeplearning4j、tensorflow 等其他平台还原。

该 HDF5 文件包含信息如下：

- 模型拓扑，可以用来重新生成模型。
- 模型已训练的权重参数。
- 模型的损失函数，优化器配置等超参数。
- 优化器状态，可以用来从上次训练点还原并继续训练。

保存了 HDF5 文件后，用户可使用 keras.models.load_model(filepath) 回复模型并实例化。实例化时 load_model 方法还完成了模型的编译，保

证模型的前期配置都还原完毕（除非前期该模型未被编译）。

以下是示例代码：

```
from keras.models import load_model

model.save('my_model.h5')
del model

model = load_model('my_model.h5')
```

上述代码完成了一个模型从备份到销毁再到还原的完整过程，最后实例化的 model 就是在上一次备份点的模型。

如果要保存模型拓扑，并不关心已训练参数状态、超参数等，可以仅仅使用 to_json 或者 to_yaml 方法。

```
json_string = model.to_json()
yaml_string = model.to_yaml()
```

这两个方法会把模型拓扑保存成 json 或者 yaml 字符串。

同时，Keras 提供了拓扑还原接口：

```
from keras.models import model_from_json
model = model_from_json(json_string)

from keras.models import model_from_yaml
model = model_from_yaml(yaml_string)
```

有时，只是想要保存模型已训练的权重参数，这时可以使用 save_weights 方法：

```
model.save_weights('my_model_weights.h5')
```

并且可以还原模型参数（前提是模型拓扑是一致的）：

```
model.load_weights('my_model_weights.h5')
```

如果模型拓扑不一致，只是想要尽量通过网络层的名字还原权重参数，需要指定 by_name=True。

```
model.load_weights('my_model_weights.h5', by_name=True)
```

以下是示例代码。

假定原始模型拓扑如下：

```
model = Sequential()
model.add(Dense(2, input_dim=3, name="dense_1"))
model.add(Dense(3, name="dense_2"))
...
model.save_weights(fname)
```

可以通过如下方法，尽可能地通过网络层的名字还原：

```
model = Sequential()
model.add(Dense(2, input_dim=3, name="dense_1"))
model.add(Dense(10, name="new_dense"))

model.load_weights(fname, by_name=True)
```

8.2　Keras 内置 Scikit-Learn 接口包装器

Scikit-Learn 机器学习库的简约易用与 Keras 框架非常相似。如果读者曾经是 Scikit-Learn 的用户，并且想要在 Keras 中使用 Scikit-Learn 的一些特有功能（如交叉验证、网格搜索等），可以使用 Keras 内置的 Scikit-Learn 接口包装器。

通过包装器将 Sequential 模型（仅有一个输入）作为 Scikit-Learn 工

作流的一部分，相关的包装器定义在 keras.wrappers.scikit_learn.py 中。

目前，有两个包装器可用：

```
keras.wrappers.scikit_learn.KerasClassifier(build_fn=None,
**sk_params)
```

实现了 sklearn 的分类器接口：

```
keras.wrappers.scikit_learn.KerasRegressor(build_fn=None,
**sk_params)
```

实现了 sklearn 的回归器接口：

参数如下：

- build_fn：可调用的函数或类对象。

  build_fn 应构造、编译并返回一个 Keras 模型。该模型将稍后用于训练/测试。build_fn 的值可能为下列 3 种之一：

  ✧ 一个函数。

  ✧ 一个具有 call 方法的类实例。

  ✧ None，代表类继承自 KerasClassifier 或 KerasRegressor，其 call 方法是其父类的 call 方法。

- sk_params：模型参数和训练参数。

  sk_params 合法的模型参数为 build_fn 的参数。注意，“build_fn”应提供其参数的默认值。所以我们不传递任何值给 sk_params 也可以创建一个分类器/回归器。sk_params 还接受用于调用 fit、predict、

predict_proba 和 score 方法的参数，如 nb_epoch、batch_size 等。这些用于训练或预测的参数按如下先后顺序选择：

- ✧ 传递给 fit、predict、predict_proba 和 score 的字典参数。
- ✧ 传递给 sk_params 的参数。
- ✧ keras.models.Sequential、fit、predict、predict_proba 和 score 的默认值。

当使用 scikit-learn 的网格搜索（grid_search）接口时，合法的可调参数是可以传递给 sk_params 的参数，包括训练参数。即可以使用 grid_search 来搜索最佳的 batch_size 或 nb_epoch 以及其他模型参数。

自动超参数优化是 Scikit-Learn 接口中非常实用的功能之一。这种优化方式无需人工调参，其中最常用的两种方法是网格搜索（grid_search）和随机化搜索（randomized_search），如图 8-1 所示。网格搜索会尝试给出的所有超参数组合；而随机化搜索更多地使用在超参数较多的情况，会在超参数空间的一个特定分布上随机抽样一些超参数组合，最后返回抽样到的最好超参数组合。

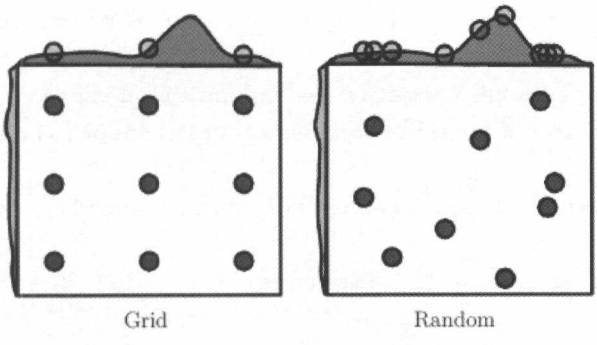


图 8-1　网格搜索和随机化搜索比较

下面展示了如何使用 Scikit-Learn 的网格搜索（grid_search）自动寻找合适超参数的过程。

整个任务是建立一个简单的卷积网络识别 mnist 手写数字，并且用网格搜索方法自动调优一些超参数，返回最优超参数。首先，代码中我们导入一些必须的库，包括前面提到的 Keras 包装器 KerasClassifier 和 Scikit-Learn 的 GridSearchCV 方法。

```
    from __future__ import print_function

    import keras
    from keras.datasets import mnist
    from keras.models import Sequential
    from keras.layers import Dense, Dropout, Activation, Flatten
    from keras.layers import Conv2D, MaxPooling2D
    from keras.wrappers.scikit_learn import KerasClassifier
    from keras import backend as K
    from sklearn.grid_search import GridSearchCV


    num_classes = 10

    img_rows, img_cols = 28, 28

    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    if K.image_data_format() == 'channels_first':
        x_train = x_train.reshape(x_train.shape[0], 1, img_rows,
img_cols)
        x_test = x_test.reshape(x_test.shape[0], 1, img_rows,
img_cols)
        input_shape = (1, img_rows, img_cols)
    else:
        x_train = x_train.reshape(x_train.shape[0], img_rows,
```

```
img_cols, 1)
        x_test = x_test.reshape(x_test.shape[0], img_rows,
img_cols, 1)
        input_shape = (img_rows, img_cols, 1)

    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255

    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    def make_model(dense_layer_sizes, filters, kernel_size,
pool_size):
        '''Creates model comprised of 2 convolutional layers
followed by dense layers

        dense_layer_sizes: List of layer sizes.
            This list has one number for each layer
        filters: Number of convolutional filters in each
convolutional layer
        kernel_size: Convolutional kernel size
        pool_size: Size of pooling area for max pooling
        '''

        model = Sequential()
        model.add(Conv2D(filters, kernel_size,
                         padding='valid',
                         input_shape=input_shape))
        model.add(Activation('relu'))
        model.add(Conv2D(filters, kernel_size))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=pool_size))
        model.add(Dropout(0.25))

        model.add(Flatten())
```

```
    for layer_size in dense_layer_sizes:
        model.add(Dense(layer_size))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes))
    model.add(Activation('softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer='adadelta',
                  metrics=['accuracy'])

    return model
```

我们构造一个简单的有二层卷积的网络，注意我们把全连接层留空，假设需要添加的全连接层的数量和输出未知。即 dense_layer_sizes 是我们想要调优的超参数，我们想要考虑这样一些情况。

（1）再添加 1 层 32 输出的全连接层。

（2）再添加 1 层 64 输出的全连接层。

（3）再添加 2 层 32 输出的全连接层。

（4）再添加 2 层 64 输出的全连接层。

（5）先添加 1 层 32 输出的全连接层，再添加 1 层 64 输出的全连接层。

（6）先添加 1 层 64 输出的全连接层，再添加 1 层 32 输出的全连接层。

因此，所有情况列表为[[32], [64], [32, 32], [64, 64], [32, 64], [64, 32]]，根据这些情况我们进行网格搜索。通过 Keras 的 KerasClassifier 接口获得经过包装的 Scikit-Learn 模型实例，把此模型实例放到 Scikit-Learn 的网格搜索 GridSearchCV 接口中进行模型的超参数搜索和验证。

```
    dense_size_candidates = [[32], [64], [32, 32], [64, 64], [32,
64], [64, 32]]
    my_classifier = KerasClassifier(make_model, batch_size=32)
    validator = GridSearchCV(my_classifier,
                             param_grid={'dense_layer_sizes':
dense_size_candidates,
                                         'epochs': [3, 6],
                                         'filters': [8],
                                         'kernel_size': [3],
                                         'pool_size': [2]},
                             scoring='neg_log_loss',
                             n_jobs=1)
```

注意，虽然 epochs、filters、kernel_size、pool_size 等参数没有在构造模型时明确指定，这些参数默认在 Scikit-Learn 接口中是可调超参数，可以直接在使用 GridSearchCV 时指定调优的组合。

```
    validator.fit(x_train, y_train)
    print('The parameters of the best model are: ')
    print(validator.best_params_)
```

最后，调用.fit 方法进行超参数搜索，输出结果。此处 validator.best_estimator_ 返回的是被 Keras 包装过的 Scikit-Learn 模型对象，validator.best_estimator_.model 返回的是未包装的 Keras 模型对象。

```
    best_model = validator.best_estimator_.model
    metric_names = best_model.metrics_names
    metric_values = best_model.evaluate(x_test, y_test)
    for metric, value in zip(metric_names, metric_values):
        print(metric, ': ', value)
```

输出如下：

```
    The parameters of the best model are:
    {'epochs': 6, 'dense_layer_sizes': [64, 64], 'kernel_size':
```

```
3, 'filters': 8, 'pool_size': 2}
    acc :  0.9815
```

经过搜索，可以得出结论，最好的超参数组合是“再添加 2 层 64 输出的全连接层”。

8.3 Keras 内置可视化工具

Keras 内置可视化工具 keras.utils.vis_utils 模块提供了生成 Keras 模型拓扑图的函数（使用 graphviz）。该函数将画出模型拓扑结构图，并保存成图片。

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

plot_model 接收两个可选参数。

- show_shapes：用于指定是否显示输出数据的形状，默认为 False。
- show_layer_names：用于指定是否显示层名称，默认为 True。

输出图片 model.png 的样例如图 8-2 所示。

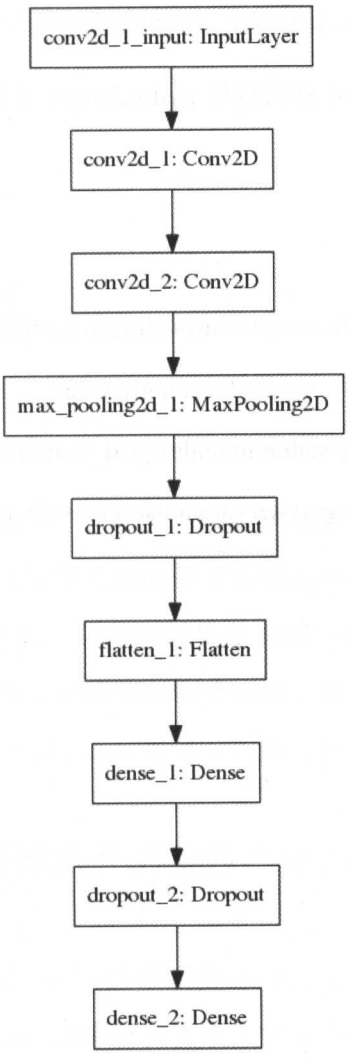


图 8-2　Keras 项目自带例程中 mnist_cnn 的拓扑可视化

我们也可以直接获取一个 pydot.Graph 对象，然后按照自己的需要渲染它，例如，如果要在 ipython 中展示图片，可以尝试下述代码：

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
```

```
SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

其中，model_to_dot 方法得到 pydot.Graph 对象，create 方法渲染想要输出的格式。

参考文献

[1] https://groups.google.com/forum/#!forum/keras-users

[2] https://deeplearning4j.org/model-import-keras

[3] http://scikit-learn.org/stable/modules/grid_search.html#grid-search-tips

[4] http://machinelearningmastery.com/use-keras-deep-learning-models-scikit-learn-python/

第 9 章

Keras 实战

Keras 是一个简约实用，适合作为深度学习原型孵化的框架。通过前面章节的学习，我们已经掌握充分的知识去实现一个完整训练任务。借助 Keras 快速构建原型的优势，我们可以快速实现一些性能提升测试，新型网络架构验证，以及一些应用方案的初步试验。本章我们完成一些 Keras 实战练习，增强读者对完整 Keras 训练过程的理解。

9.1 训练一个准确率高于 90%的 Cifar-10 预测模型

本节我们训练一个 Cifar-10 的预测模型。为了达到较高性能与准确率，我们借鉴和还原 ICLR 2016 的来自捷克理工大学论文《ALL YOU NEED IS A GOOD INIT》。我们训练一个类似 VGG 的包含 15 层卷积的深度网络，使用论文提到的 LSUV 权重初始化方法和 Keras 自带强大的图像增强预处理进行训练。

本节使用的数据集 Cifar-10 是深度学习中经典的基准数据集，并经常用来测试深度网络性能指标。Cifar-10 数据集是最初由深度学习之父

Geoffrey Hinton，以及他的两名学生 Alex Krizhevsky 和 Vinod Nair 从 80 million tiny images 整理出来的子集。正如其名，Cifar-10 数据集包含 10 个类别，分别为 airplane、automobile、bird、cat、deer、dog、frog、horse、ship 和 truck，如图 9-1 所示。其中每个类别包含 6000 个样本，因此整个数据集包含 60 000 张图片，其中训练集 50 000 张，测试集 10 000 张。每张图片是严格标注分类的 32 × 32 的彩图，因此不会同时出现在两个类别中。

图 9-1　Cifar-10 数据集示例

目前许多公认的研究与应用都会基于 Cifar-10 数据集做性能测试。目前 state-of-the-art 的研究工作已经可以达到 96.53%的正确率。本节使用的主要 Tricks 是特殊的权重初始化方法和 Keras 自带的强大的图像增强工具。但是，如果需达到高于 90%准确率，还需要训练较长时间。经测试，使用英伟达 Tesla K40m 用时大约 200～300 个 epoch（5 小时训练时间）

即可达到 90%的准确率。随着训练时间的增加和持续的图像增强，准确率还会有所提升。

本例使用的 Cifar-10 预测模型整体拓扑如下：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
activation_3 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 48)        13872
_________________________________________________________________
activation_4 (Activation)    (None, 32, 32, 48)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 48)        20784
_________________________________________________________________
activation_5 (Activation)    (None, 32, 32, 48)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 48)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 16, 48)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 16, 16, 80)        34640
_________________________________________________________________
activation_6 (Activation)    (None, 16, 16, 80)        0
_________________________________________________________________
```

```
conv2d_7 (Conv2D)            (None, 16, 16, 80)        57680
_________________________________________________________________
activation_7 (Activation)    (None, 16, 16, 80)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 16, 80)        57680
_________________________________________________________________
activation_8 (Activation)    (None, 16, 16, 80)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 80)        57680
_________________________________________________________________
activation_9 (Activation)    (None, 16, 16, 80)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 80)        57680
_________________________________________________________________
activation_10 (Activation)   (None, 16, 16, 80)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 80)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 8, 8, 80)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 8, 8, 128)         92288
_________________________________________________________________
activation_11 (Activation)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
activation_12 (Activation)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
activation_13 (Activation)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
activation_14 (Activation)   (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 8, 8, 128)         147584
```

```
activation_15 (Activation)   (None, 8, 8, 128)          0
_________________________________________________________________
global_max_pooling2d_1 (Glob (None, 128)                0
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)                0
_________________________________________________________________
dense_1 (Dense)              (None, 500)                64500
_________________________________________________________________
activation_16 (Activation)   (None, 500)                0
_________________________________________________________________
dropout_4 (Dropout)          (None, 500)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                 5010
_________________________________________________________________
activation_17 (Activation)   (None, 10)                 0
=================================================================
Total params: 1,071,542
Trainable params: 1,071,542
Non-trainable params: 0
_________________________________________________________________
```

这是一个类似 VGG 的深度网络，总共 15 个的卷积层，每 5 个卷积层后有一个池化层，即总共 3 个池化层，最后设置了两层全连接层（一个输出为 500 个神经元，另一个为 10 个神经元）和一个 softmax 归一化层。另外，所有卷积层后都添加了 Relu 激活层.为了抵制过拟合，所有池化层后都添加 Dropout 层。在进行完整的训练过程之前，我们首先下载之前提到的 LSUV 初始化方法库，提高模型性能。

```
git clone https://github.com/ducha-aiki/LSUV-keras
```

把此库的 py 文件复制到当前工作目录，就可以开始下一步的数据预处理和训练工作了。

9.1.1 数据预处理

对于标准数据集 Cifar-10，Keras 内置数据集中已经有提供接口，因此我们可以跳过数据收集和清洗的过程。我们首先需要导入数据集，以及接下来训练需要的 Keras 网络层（如全连接层、2D 卷积层、2D 池化层和 Dropout 层等）。对于图像预增强，我们要使用 Keras 提供的强大接口 ImageDataGenerator，并且导入 LSUVinit 初始化方法库。

```
    from __future__ import print_function
    import keras
    from keras.datasets import cifar10
    from keras.preprocessing.image import ImageDataGenerator
    from keras.models import Sequential
    from keras.layers import Dense, Dropout, Activation, Flatten
    from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D,
GlobalMaxPooling2D
    from lsuv_init import LSUVinit
```

然后，我们设置训练的全局超参数。batch_size 设置为 32，即每步训练同时使用 32 个样本。Cifar-10 类别总数为 10，所以 num_classes=10。总的训练 epochs 设为 1600，一次 epoch 即所有训练集都训练一次。如果不确定网络什么时候大致收敛，可以选择大一些的 epoch 数。并且，如果使用了图像增强并且希望训练结果更好，可以适当增加 epoch 数，因为数据的持续增强可以保证训练结果更久的持续提升。在这次训练中，我们默认使用数据增强，即 data_augmentation = True。

```
    batch_size = 32
    num_classes = 10
    epochs = 1600
    data_augmentation = True
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

当我们使用 Keras 自带的加载数据接口 cifar10.load_data()，实际得到的数据是已经随机打乱过的。最后的数据预处理我们把所有的样本标签用 to_categorical 设置成总类别数长度的数组，如第 2 类的标签为 [0,1,0,0,0,0,0,0,0,0]，便于训练计算。

9.1.2　训练

训练前我们必须构造事先设计的深度网络模型，首先得到实例化的贯序模型 model。之后根据设计加入需要的网络层，注意卷积层有 padding 参数，如果设置为“same”，则可以在图片外围补齐缺失的像素（默认为添 0 补齐），控制该卷积层输出的特征图像大小，这在快速构建原型和为下一层准备特定大小输入的时候非常有用。另外，类似 VGG 网络，我们每隔 5 层加入了 2 ×2 的池化层，用来减少计算量并抵制过拟合。Dropout 数量和参数我们可以自行调节。如果 Dropout 参数增大，即以更大的概率抑制与下层的神经元连接，可以抵制过拟合，但容易产生欠拟合的情况；如果 Dropout 参数减小，即以更小的概率抑制与下一层的神经元连接，越容易产生过拟合的情况。

```
model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
```

```
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(48, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(48, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(80, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(80, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(80, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(80, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(80, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(128, (3, 3), padding='same',
                     input_shape=x_train.shape[1:]))
    model.add(Activation('relu'))
```

```
model.add(Conv2D(128, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(GlobalMaxPooling2D())
model.add(Dropout(0.25))

model.add(Dense(500))
model.add(Activation('relu'))
model.add(Dropout(0.4))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

opt = keras.optimizers.Adam(lr=0.0001, decay=1e-6)
```

至此，我们已经把所有网络拓扑构建完成。剩下很重要的是选择我们的训练“引擎”：优化器。我们选择各方面表现较好的 Adam 优化器，学习率选择建议的默认值 0.0001，因为我们希望训练完全并且使用较长时间，为了在整个训练时间内没有太大过拟合情况，我们设置学习率衰减为 1e-6，希望在较多训练步长后，使用较小步长更缓和，而不是“激进”地训练，从而获得更好的收敛值。

模型编译阶段，我们使用常用的类别交叉熵损失函数，指定之前的 Adam 优化器，以及 Keras 内置的准确率性能评估函数，保证训练时得到模型基本的性能评估情况。

```
    model.compile(loss='categorical_crossentropy',
                  optimizer=opt,
                  metrics=['accuracy'])


    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255

    model = LSUVinit(model,x_train[:batch_size,:,:,:])
    tbCallBack = keras.callbacks.TensorBoard(log_dir='./Graph',
histogram_freq=0, write_graph=True, write_images=True)
```

这里我们用 LSUVinit 类初始化模型待训练参数。另外，我们在训练前指定额外的 Keras 内置 TensorBoard 回调函数，保证训练过程中实时对性能评估进行可视化，方便调试。

实际训练阶段，我们使用 Keras 内置图像增强的预处理工具，使用以下一些变换：

- 对图像进行不超过 10° 的随机旋转（rotation_range=10）。
- 随机水平平移图片 20%（width_shift_range=0.2）。
- 随机垂直平移图片 20%（width_shift_range=0.2）。
- 随机翻转图片（horizontal_flip=True）。

```
    if not data_augmentation:
        print('Not using data augmentation.')
        model.fit(x_train, y_train,
                  batch_size=batch_size,
                  epochs=epochs,
                  validation_data=(x_test, y_test),
```

```
                shuffle=True, callbacks=[tbCallBack])
    else:
        print('Using real-time data augmentation.')
        datagen = ImageDataGenerator(
            featurewise_center=False,
            samplewise_center=False,
            featurewise_std_normalization=False,
            samplewise_std_normalization=False,
            zca_whitening=False,
            rotation_range=10,
            width_shift_range=0.2,
            height_shift_range=0.2,
            horizontal_flip=True,
            vertical_flip=False)

        datagen.fit(x_train)

        model.fit_generator(datagen.flow(x_train, y_train,
                            batch_size=batch_size),
                steps_per_epoch=x_train.shape[0] // batch_size,
                    epochs=epochs,
                validation_data=(x_test,  y_test),  callbacks=
[tbCallBack])
```

最后，调用.fit 或者.fit_generator 方法开始训练。如果调用.fit 方法，实际是静态训练，无法使用 Keras 提供的实时数据增强功能。如果调用.fit_generator 方法，则训练数据可以传入 datagen.flow 返回的无限循环迭代器，无限生成经过增强的图像。至此我们已经启动了训练。

在终端可以看到 Keras 内置的进度条显示实时训练数据。因为我们在训练时也指定了 TensorBoard 回调函数 tbCallBack，因此我们也可以使用以下命令指定 TensorBoard log 目录并启动 TensorBoard：

```
tensorboard --logdir=Graph --host 0.0.0.0 --port 6006
```

其中，Graph 正是我们之前在回调定义时指定的目录：

```
    tbCallBack = keras.callbacks.TensorBoard(log_dir='./Graph',
histogram_freq=0, write_graph=True, write_images=True)
```

这样启动 TensorBoard 服务后，我们可以访问 6006 端口，观察训练情况。

图 9-2 是使用英伟达 Tesla K40m 训练结果，用时大约 200～300 个 epoch（5 小时训练时间）达到 90%的准确率，持续训练到 1600 个 epoch 可达到 91%以上的准确率。如果读者对提高准确率有额外的兴趣，可继续自行调节超参数或采用自动超参数优化算法。因为本例程没有使用太多其他的 Tricks，读者也可以尝试其他新兴或成熟的 Tricks，如 Batch Normalization，Fractional Max-Pooling，或其他激活函数。

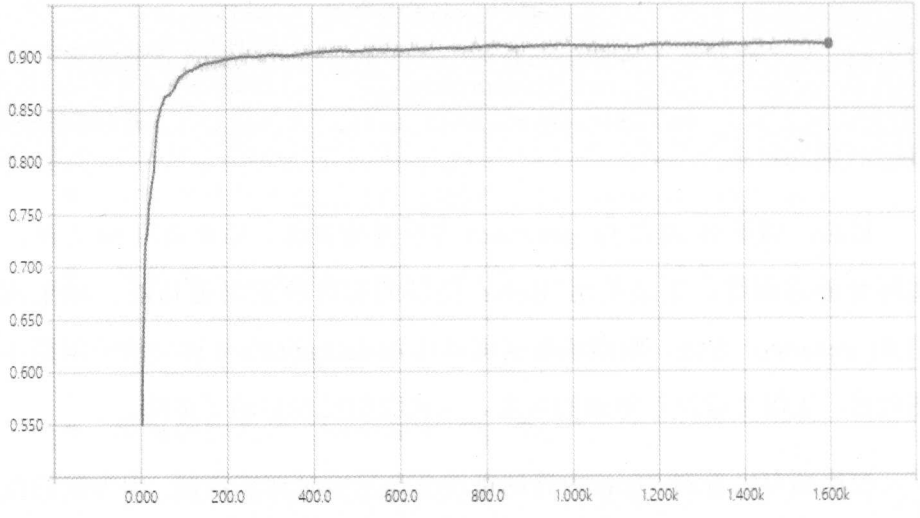


图 9-2　Tensorboard 记录的测试集准确率变化

9.2　在 Keras 模型中使用预训练词向量判定文本类别

在自然语言处理中，“词向量”（又称为词嵌入）是将一类将词的语义映射到向量空间中去的技术。正如其名，它将一个词用特定的向量编码表示，向量之间的距离（例如，任意两个向量之间的 L2 范式距离或更常用的余弦距离）在一定程度上表征了的词之间的语义关系。由这些向量形成的几何空间被称为一个嵌入空间。如图 9-3 所示，是单词映射到向量空间（嵌入空间）后在“男性-女性”“动词”“国家-首都”3 种关系中的语义表征。

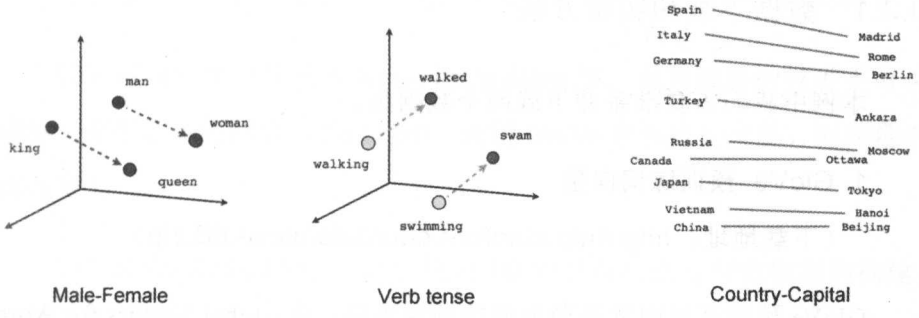


图 9-3　单词映射到词向量空间中的语义表征

例如，“king”和“woman”是语义上很难联系到一起的词，所以它们的词向量在一个合理的嵌入空间的距离将会非常遥远。但“king”和“man”有一定的相关性（king 一般是 man），所以它们的词向量之间的距离会相对小。

理想的情况下，在一个良好的嵌入空间里，从“king”向量到“man”向量的“路径”向量会精确地捕捉这两个概念之间的语义关系。在这种情况下，“路径”向量表示的是“统治的属性”（king 如果去除性别的属性，

最大的特征是有统治的属性），所以期望“king”向量 - “man”向量（两个词向量的差异）捕捉到“统治的属性”这样的语义关系。基本上，我们应该有向量等式：king – man + woman = queen。如果真的是如此，那么我们可以使用这样的关系向量来回答某些问题。例如，应用这种语义关系到一个新的向量，比如“工作”，我们应该得到一个有意义的等式，工作+ 发生的地点 = 办公室，来回答“工作发生在哪里？”。

词向量通过降维技术表征在文本预料库中单词共现的信息。词向量方法包括神经网络（通常所说的“Word2vec”技术），或矩阵分解。

9.2.1 数据下载和实验方法

本例中我们需要准备并下载两个数据集。

1. GloVe 预训练词向量

（下载地址：http://nlp.stanford.edu/data/glove.6B.zip）

GloVe 预训练词向量是斯坦福的开源项目，是 Global Vectors for Word Representation 的缩写，一种基于共现矩阵分解的词向量。本节所使用的预训练 GloVe 词向量是在 2014 年的英文维基百科上训练的，有 40 万个不同的单词，每个单词用 100 维向量表示。注意，词向量文件大小约为 822MB，请确定网络通畅。最后把数据存放在“/glove.6B/”目录下。

2. 20 Newsgroup dataset

（20 个新闻组数据集，下载地址：http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.html）

本例的判别任务需要识别的是著名的“20 Newsgroup dataset”。该数

据集共有 20 种新闻文本数据，我们将实现对该数据集的文本分类任务。最后把数据存放在“/20_newsgroup/”目录下。

不同类别的新闻预料包含大量不同的单词，在语义上存在很大的差别。

以下是我们的实验步骤：

（1）将所有的新闻样本转化为词索引序列，即为每一个词依次分配一个整数 ID。遍历所有的新闻文本，我们只保留最常见的 20 000 个词，而且，每个新闻文本最多保留 1000 个单词。

（2）生成一个词向量矩阵。第 i 列表示词索引为 i 的单词的词向量。

（3）将词向量矩阵载入 Keras Embedding 层，设置该层的权重不可再训练（即在之后的网络训练过程中，使用 GloVe 预训练词向量，词向量不再改变）。

（4）Keras Embedding 层之后使用 1D 的卷积层进行判定模型的构建，并用一个 softmax 全连接层输出新闻类别判断。

9.2.2　数据预处理

首先，我们导入所有必要的库，定义需要的全局变量，并且把所有需要的数据集载入内存对象。其中，MAX_SEQUENCE_LENGTH 是每个新闻文本最多保留的单词数 1000，MAX_NB_WORDS 是所有新闻样本总的字典中单词数，EMBEDDING_DIM 是词嵌入层的输出尺寸 100，VALIDATION_SPLIT 代表测试集占到总数据集的比例 0.2。

处理数据阶段，首先，我们从 GloVe 文件中解析出每个词和它所对应

的词向量，并用字典的方式存储在对象 embeddings_index 中。

```
from __future__ import print_function

import os
import sys
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.layers import Dense, Dropout, Input, Flatten
from keras.layers import Conv1D, MaxPooling1D, Embedding
from keras.models import Model


BASE_DIR = ''
GLOVE_DIR = BASE_DIR + '/glove.6B/'
TEXT_DATA_DIR = BASE_DIR + '/20_newsgroup/'
MAX_SEQUENCE_LENGTH = 1000
MAX_NB_WORDS = 20000
EMBEDDING_DIM = 100
VALIDATION_SPLIT = 0.2

print('Indexing word vectors.')

embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

输出如下：

```
Indexing word vectors.
Found 400000 word vectors.
```

然后，我们遍历语料文件下的所有文件夹，获得不同类别的新闻文本 19 997 篇，以及它们对应的类别标签，所有新闻文本用序列 texts 存储，所有标签用序列 labels 存储。其代码如下所示：

```
print('Processing text dataset')

texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric
id
labels = []  # list of label ids
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                if sys.version_info < (3,):
                    f = open(fpath)
                else:
                    f = open(fpath, encoding='latin-1')
                t = f.read()
                i = t.find('\n\n')  # skip header
                if 0 < i:
                    t = t[i:]
                texts.append(t)
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
```

输出如下：

```
Processing text dataset
Found 19997 texts.
```

之后，我们可以将新闻样本转化为神经网络训练所用的张量。所用到的 Keras 库是 keras.preprocessing.text.Tokenizer 和 keras.preprocessing.sequence.pad_sequences。

代码如下所示：

```
# finally, vectorize the text samples into a 2D integer tensor
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
num_validation_samples = int(VALIDATION_SPLIT * data.shape[0])

x_train = data[:-num_validation_samples]
y_train = labels[:-num_validation_samples]
x_val = data[-num_validation_samples:]
y_val = labels[-num_validation_samples:]
```

输出如下：

```
Found 174105 unique tokens.
Shape of data tensor: (19997, 1000)
Shape of label tensor: (19997, 20)
```

此时，我们可以根据之前的 embeddings_index 预训练的词向量字典，对应得到目前所有新闻文本的字典中单词对应的词向量矩阵：

```
print('Preparing embedding matrix.')

# prepare embedding matrix
num_words = min(MAX_NB_WORDS, len(word_index))
embedding_matrix = np.zeros((num_words, EMBEDDING_DIM))
for word, i in word_index.items():
    if i >= MAX_NB_WORDS:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

9.2.3　训练

训练阶段，我们将这个词向量矩阵加载到 Embedding 层中，注意，我们设置 trainable=False，使得这个编码层不可再训练。

```
# load pre-trained word embeddings into an Embedding layer
# note that we set trainable = False so as to keep the
embeddings fixed
embedding_layer = Embedding(num_words,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],

input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

一个 Embedding 层的输入应该是一系列的整数序列，比如一个 2D 的输入，它的 shape 值为（samples, indices），也就是一个 samples 行，indeces 列的矩阵。每一次的 batch 训练的输入应该被 padded 成相同大小（尽管 Embedding 层有能力处理不定长序列，如果不指定数列长度这一参数。所有的序列中的整数都将被对应的词向量矩阵中对应的列（即它的词向量）代替，比如序列[1,2]将被序列[词向量[1],词向量[2]]代替。这样，输入一个 2D 张量后，我们可以得到一个 3D 张量。

最后，我们可以使用一个 3 层小型的 1D 卷积解决这个新闻分类问题。

```
    print('Training model.')

    # train a 1D convnet with global maxpooling
    sequence_input    =    Input(shape=(MAX_SEQUENCE_LENGTH,),
dtype='int32')
    embedded_sequences = embedding_layer(sequence_input)
    x = Conv1D(256, 5, activation='relu')(embedded_sequences)
    x = MaxPooling1D(5)(x)
    x = Dropout(0.4)(x)
    x = Conv1D(256, 5, activation='relu')(x)
    x = MaxPooling1D(5)(x)
    x = Dropout(0.4)(x)
    x = Conv1D(256, 5, activation='relu')(x)
    x = MaxPooling1D(35)(x)
    x = Dropout(0.25)(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)
    preds = Dense(len(labels_index), activation='softmax')(x)

    model = Model(sequence_input, preds)
    model.compile(loss='categorical_crossentropy',
                  optimizer='rmsprop',
                  metrics=['acc'])
```

```
model.fit(x_train, y_train,
          batch_size=64,
          epochs=10,
          validation_data=(x_val, y_val))
```

如果读者觉得训练效果不满意，可以使用自己的预训练词向量而非 GloVe 词向量，或者使用 Keras 自带的 Embedding 层进行训练，从而对比性能。

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

9.3　用 Keras 实现 DCGAN 生成对抗网络还原 MNIST 样本

本节将学习如何通过用 Keras 构建一个生成对抗网络，还原和模仿 MNIST 样本。

2014 年，Ian Goodfellow 等学者发表论文《Generative Adversarial Nets》，即《生成对抗网络》，标志着 GANs 的诞生。文中，Ian Goodfellow 等作者详细介绍了 GANs 的原理、优点，以及在图像生成方面的应用。自此，GANs 持续了飞速的发展与发展，学界、业界对 GANs 的兴趣出现“井喷”。

- 多篇重磅论文陆续发表。
- Facebook、Open AI 等 AI 业界巨头投入对 GANs 的研究。
- 它成为 2016 年 12 月 NIPS 大会当之无愧的明星——在会议大纲

中被提到逾 170 次。

- 在 2017 深度学习盛会 ICLR 大会上，GAN 相关论文提交依然非常之多。
- GANs 之父 Ian Goodfellow 被公认为人工智能的顶级专家。
- 深度学习开创者 Yan Lecun 对 GANs 交口称赞，称其为“20 年来机器学习领域最酷的想法”。

那么究竟什么是生成对抗网络？

生成对抗网络（GAN）是非监督学习的一种方法，通过让两个神经网格相互博弈的方式进行学习。本质上，GAN 目标是训练出一个好的生成模型，来模拟训练集中的数据。不同的是，一般的生成模型，必须先初始化一个“假设分布”，即后验分布，通过各种抽样方法抽样这个后验分布，就能知道这个分布与真实分布之间究竟有多大差异。这里的差异就要通过构造损失函数（loss function）来估算。知道了这个差异后，就能不断调优一开始的“假设分布”，从而不断逼近真实分布。限制玻尔兹曼机（RBM）就是这种生成模型的一种。然而，对抗网络可以学习自己的损失函数，无须精心设计和建构一个损失函数，却能达成无监督学习。

如图 9-4 所示，生成对抗网络同时训练两个模型，称为生成器（Generator）和判别器（Discriminator）。生成器目标是尽力模仿真实分布生成数据，而判别器的目标是区分出真实样本和生成器生成的模仿样本，直到判断器无法区分出真实样本和模仿样本为止。通过这种方式，损失函数被蕴含在判别器中了。我们不再需要思考损失函数应该如何设定，而只

要保证最后的生成器收敛到一个比较好的状态。

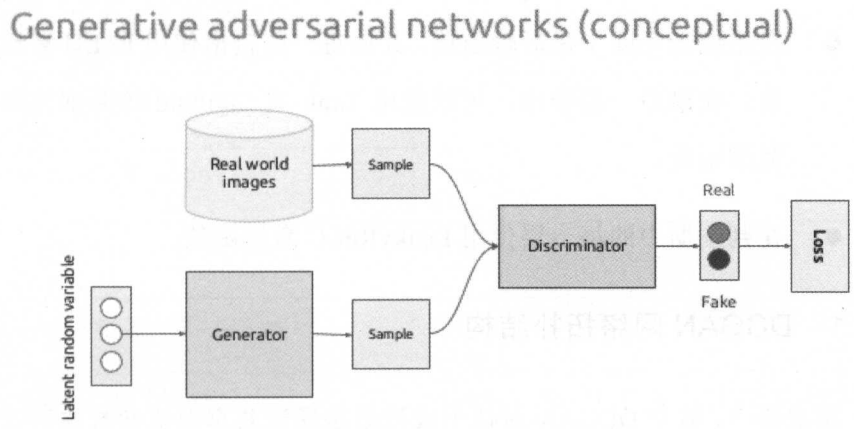


图 9-4　GAN 原理概念图

虽然，省去复杂的后验推断过程是 GANs 相对其他生成模型的优势，但是，早期的 GANs 有许多问题，最主要的一项通病是 GANs 不稳定——“有时候它永远不会开始学习，或者生成我们认为合格的输出。”深度卷积生成对抗网络（DCGAN）的提出为解决这一问题提供了很好的参考。DCGAN 提出了训练 GAN 这种不稳定模型的重要架构设计，并总结了针对 CNN 这种网络的特定经验，使得改进的 CNN 结构能够在多种数据集上稳定地训练。

DCGAN 方法对 CNN 使用和修改的核心建议如下：

- 在判别器中用带步长的卷积层（strided convolutions）取代的池化层（pooling layers）。在生成器中用小步幅卷积（fractional strided convolutions）取代的池化层（pooling layers），达到学习上采样的效果。

- 在判别器和生成器中都采用 Batch Normalization 批标准化。

- 对于较深的网络，移除全连接层。
- 在生成器中除了最后输出层，其他每一层输出使用 ReLU 激活函数。在最后一层输出，可以使用 Tanh 或 Sigmoid 等两端饱和的激活函数。
- 在判别器中的每一层使用 LeakyReLU 激活函数。

9.3.1 DCGAN 网络拓扑结构

在本例中，整个 DCGAN 对抗生成模型是通过共享训练参数的形式训练的。生成器和判别器通过共享同一组训练参数，交替更新这组参数，达到“对抗”训练的效果。训练过程中，生成器和判别器在类似博弈的游戏中，使得参数趋于训练收敛的同时，生成器的输出效果越来越好。为了直观理解，首先总览本例要训练的拓扑结构。

用于本例的拓扑结构包含 4 个 Keras 贯序对象（sequential），事实上只有两个贯序对象实际包含了待训练参数，其他两个对象只是引用前两个对象达到参数复用的效果。4 个贯序对象如下。

判别器（sequential_2）：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 14, 14, 64)        1664
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 128)         204928
_________________________________________________________________
```

```
dropout_2 (Dropout)          (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 256)         819456
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 4, 512)         3277312
_________________________________________________________________
dropout_4 (Dropout)          (None, 4, 4, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 8193
_________________________________________________________________
activation_1 (Activation)    (None, 1)                 0
=================================================================
Total params: 4,311,553
Trainable params: 4,311,553
Non-trainable params: 0
_________________________________________________________________
```

判别器贯序对象指定的 cnn 拓扑是按照 DCGAN 建议的架构。这不仅包含了输出信息中的显示的 4 层卷积网络，而且在每层卷积中使用了带步长的卷积。并且，随后在代码中可以注意到，每层卷积之后使用了 LeakyReLU，而不是传统 RelU。最后，使用了一个展平层（flatten_1），一个全连接层（dense_1）和一个 Sigmoid 激活函数完成模型输出。如果输出是“0”，则表示判定输入样本为伪造的的样本；如果输出是“1”，则表示判定输入样本为真实样本。

判别器模型（sequential_1）：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
sequential_2 (Sequential)    (None, 1)                 4311553
=================================================================
Total params: 4,311,553
Trainable params: 4,311,553
Non-trainable params: 0
_________________________________________________________________
```

可以注意到，这个贯序对象事实上是对判别器（sequential_2）的引用，拓扑结构和训练参数数量和 squential_1 完全一样。为了之后便于共享参数，我们才构造了这样一个判别器模型。

生成器（sequential_4）：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)                              (None, 12544)
1266944
_________________________________________________________________
batch_normalization_1 (Batch (None, 12544)             50176
_________________________________________________________________
activation_2 (Activation)    (None, 12544)             0
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 256)         0
_________________________________________________________________
dropout_5 (Dropout)          (None, 7, 7, 256)         0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 14, 14, 256)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 128)       819328
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 128)       512
_________________________________________________________________
activation_3 (Activation)    (None, 14, 14, 128)       0
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 28, 28, 128)       0
```

```
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 64)        204864
_________________________________________________________________
batch_normalization_3 (Batch (None, 28, 28, 64)        256
_________________________________________________________________
activation_4 (Activation)    (None, 28, 28, 64)        0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 28, 28, 32)        51232
_________________________________________________________________
batch_normalization_4 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
activation_5 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 28, 28, 1)         801
_________________________________________________________________
activation_6 (Activation)    (None, 28, 28, 1)         0
=================================================================
Total params: 2,394,241
Trainable params: 2,368,705
Non-trainable params: 25,536
_________________________________________________________________
```

生成器贯序对象也是依据 DCGAN 建议的生成器架构。我们使用 Keras 中的上抽样层，转置卷积层（有时称为反卷积）和批标准化层来构造传统的卷积网络。传统的卷积网络中，每一层卷积是对样本做了抽象编码，输出数据尺寸一般小于输入数据尺寸。而转置卷积（有时称为反卷积），是一个卷积层的逆行为，每一层转置卷积试图把抽象的数据编码还原成实际的样本，因此输出数据尺寸一般大于输入数据尺寸。从更大的尺度看生成器，整个贯序对象的输入尺寸是 100，输出尺寸是 28×28，即 784，把随机 100 位的张量还原成与 MNIST 非常相似的数字样本，这正是整个生成器的最终目标。

对抗模型（sequential_3）：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_4 (Sequential)          (None, 28, 28, 1)
2394241
_________________________________________________________________
sequential_2 (Sequential)    (None, 1)                 4311553
=================================================================
Total params: 6,705,794
Trainable params: 6,680,258
Non-trainable params: 25,536
```

对抗模型的贯序对象同时引用了生成器（sequential_4）和判别器（sequential_2）。这样的定义下，对抗模型（sequential_3）和判别器模型（sequential_1）就能够共享判别器（sequential_2）的拓扑和训练参数。实际的交替训练时，是由对抗模型（sequential_3）和判别器模型（sequential_1）完成交替更新参数。

9.3.2 训练

训练阶段，首先我们导入需要的 MNIST 数据集、激活函数、网络层和优化器等必要的函数模块，以及用于训练阶段保存生成图像的 matplotlib 作图工具库。随后我们构建一个简单的工具类 ElapsedTimer 用于记录总体训练经过的时间。

```
import numpy as np
import time
from tensorflow.examples.tutorials.mnist import input_data

from keras.models import Sequential
```

```
    from keras.layers import Dense, Activation, Flatten, Reshape
    from   keras.layers   import   Conv2D,   Conv2DTranspose,
UpSampling2D
    from keras.layers import LeakyReLU, Dropout
    from keras.layers import BatchNormalization
    from keras.optimizers import Adam, RMSprop

    import matplotlib.pyplot as plt

    class ElapsedTimer(object):
        def __init__(self):
            self.start_time = time.time()
        def elapsed(self,sec):
            if sec < 60:
                return str(sec) + " sec"
            elif sec < (60 * 60):
                return str(sec / 60) + " min"
            else:
                return str(sec / (60 * 60)) + " hr"
        def elapsed_time(self):
            print("Elapsed: %s " % self.elapsed(time.time() -
self.start_time) )
```

接下来我们构造深度卷积生成对抗网络的拓扑结构类 DCGAN，此处的结构与之前提到的拓扑结构一致。

类属性包括以下几个。

- Img_rows：生成图像的长度，默认为 28 像素。
- Img_cols：生成图像的宽度，默认为 28 像素。
- channel：生成图像彩色通道数，默认为 1。
- D：对应之前提到的判别器（sequential_2）。

- G：对应之前提到的生成器（sequential_4）。
- AM：对应之前提到的对抗模型（sequential_3）。
- DM：对应之前提到的判别器模型（sequential_1）。

类方法包括 4 个：discriminator()、generator()、adversarial_model()和 discriminator_model()，分别对应返回上述 D，G，AM，DM 这 4 个类属性，构造 4 个 Keras 贯序模型。

```
    class DCGAN(object):
        def __init__(self, img_rows=28, img_cols=28, channel=1):

            self.img_rows = img_rows
            self.img_cols = img_cols
            self.channel = channel
            self.D = None
            self.G = None
            self.AM = None
            self.DM = None

        def discriminator(self):
            if self.D:
                return self.D
            self.D = Sequential()
            depth = 64
            dropout = 0.4
            # In: 28 x 28 x 1, depth = 1
            # Out: 10 x 10 x 1, depth=64
            input_shape = (self.img_rows, self.img_cols, self.
channel)
            self.D.add(Conv2D(depth*1, 5, strides=2, input_shape
=input_shape,\
                padding='same',
activation=LeakyReLU(alpha=0.2)))
```

```
            self.D.add(Dropout(dropout))

            self.D.add(Conv2D(depth*2, 5, strides=2, padding=
'same',\
                    activation=LeakyReLU(alpha=0.2)))
            self.D.add(Dropout(dropout))

            self.D.add(Conv2D(depth*4, 5, strides=2, padding=
'same',\
                    activation=LeakyReLU(alpha=0.2)))
            self.D.add(Dropout(dropout))

            self.D.add(Conv2D(depth*8, 5, strides=1, padding=
'same',\
                    activation=LeakyReLU(alpha=0.2)))
            self.D.add(Dropout(dropout))

            self.D.add(Flatten())
            self.D.add(Dense(1))
            self.D.add(Activation('sigmoid'))
            self.D.summary()
            return self.D

        def generator(self):
            if self.G:
                return self.G
            self.G = Sequential()
            dropout = 0.4
            depth = 64+64+64+64
            dim = 7
            # In: 100
            # Out: dim x dim x depth
            self.G.add(Dense(dim*dim*depth, input_dim=100))
            self.G.add(BatchNormalization(momentum=0.9))
            self.G.add(Activation('relu'))
            self.G.add(Reshape((dim, dim, depth)))
            self.G.add(Dropout(dropout))
```

```
            # In: dim x dim x depth
            # Out: 2*dim x 2*dim x depth/2
            self.G.add(UpSampling2D())
            self.G.add(Conv2DTranspose(int(depth/2), 5, padding
='same'))
            self.G.add(BatchNormalization(momentum=0.9))
            self.G.add(Activation('relu'))

            self.G.add(UpSampling2D())
            self.G.add(Conv2DTranspose(int(depth/4), 5, padding
='same'))
            self.G.add(BatchNormalization(momentum=0.9))
            self.G.add(Activation('relu'))

            self.G.add(Conv2DTranspose(int(depth/8), 5, padding
='same'))
            self.G.add(BatchNormalization(momentum=0.9))
            self.G.add(Activation('relu'))

            # Out: 28 x 28 x 1 grayscale image [0.0,1.0] per pix
            self.G.add(Conv2DTranspose(1, 5, padding='same'))
            self.G.add(Activation('sigmoid'))
            self.G.summary()
            return self.G

        def discriminator_model(self):
            if self.DM:
                return self.DM
            optimizer = RMSprop(lr=0.0008, clipvalue=1.0, decay
=6e-8)
            self.DM = Sequential()
            self.DM.add(self.discriminator())
            self.DM.compile(loss='binary_crossentropy',
optimizer=optimizer,\
                metrics=['accuracy'])
            return self.DM
```

```
        def adversarial_model(self):
            if self.AM:
                return self.AM
            optimizer = RMSprop(lr=0.0004, clipvalue=1.0, decay
=3e-8)
            self.AM = Sequential()
            self.AM.add(self.generator())
            self.AM.add(self.discriminator())
            self.AM.compile(loss='binary_crossentropy',
optimizer=optimizer,\
                metrics=['accuracy'])
            return self.AM
```

所有的 DCGAN 的 Keras 拓扑结构构造完毕，剩下的工作是使用已构造的拓扑实际生成 MNIST 手写数字。因此我们需要构造类 MNIST_DCGAN 完成实际训练，显然 MNIST_DCGAN 类属性应该与训练直接相关。

类属性包括以下几个。

- Img_rows：生成图像的长度，默认为 28 像素。
- Img_cols：生成图像的宽度，默认为 28 像素。
- channel：生成图像彩色通道数，默认为 1。
- x_train：所有的训练图像，来自 MNIST 实际数据。DCGAN：之前构建的 DCGAN 类。
- discriminator：对应之前提到的判别器模型（sequential_1），即 DM。

- generator：对应之前提到的生成器（sequential_4），即 G。
- adversarial：对应之前提到的对抗模型（sequential_3），即 AM。

```
    class MNIST_DCGAN(object):
        def __init__(self):
            self.img_rows = 28
            self.img_cols = 28
            self.channel = 1

            self.x_train = input_data.read_data_sets("mnist",\
                    one_hot=True).train.images
            self.x_train = self.x_train.reshape(-1, self.img_
rows,\
                    self.img_cols, 1).astype(np.float32)

            self.DCGAN = DCGAN()
            self.discriminator = self.DCGAN.discriminator_model()
            self.adversarial = self.DCGAN.adversarial_model()
            self.generator = self.DCGAN.generator()

        def train(self, train_steps=2000, batch_size=256,
save_interval=0):
            noise_input = None
            if save_interval>0:
                noise_input = np.random.uniform(-1.0, 1.0,
size=[16, 100])
            for i in range(train_steps):
                images_train                                =
self.x_train[np.random.randint(0,
                    self.x_train.shape[0],
size=batch_size), :, :, :]
                noise = np.random.uniform(-1.0, 1.0, size=
[batch_size, 100])
                images_fake = self.generator.predict(noise)
                x = np.concatenate((images_train, images_fake))
```

```
            y = np.ones([2*batch_size, 1])
            y[batch_size:, :] = 0
            d_loss = self.discriminator.train_on_batch(x, y)

            y = np.ones([batch_size, 1])
            noise = np.random.uniform(-1.0, 1.0, size=
[batch_size, 100])
            a_loss = self.adversarial.train_on_batch(noise,
y)
            log_mesg = "%d: [D loss: %f, acc: %f]" % (i,
d_loss[0], d_loss[1])
            log_mesg = "%s  [A loss: %f, acc: %f]" % (log_mesg,
a_loss[0], a_loss[1])
            print(log_mesg)
            if save_interval>0:
                if (i+1)%save_interval==0:
                    self.plot_images(save2file=True,
samples=noise_input.shape[0],\
                        noise=noise_input, step=(i+1))

    def plot_images(self, save2file=False, fake=True,
samples=16, noise=None, step=0):
        filename = 'mnist.png'
        if fake:
            if noise is None:
                noise = np.random.uniform(-1.0, 1.0, size=
[samples, 100])
            else:
                filename = "mnist_%d.png" % step
            images = self.generator.predict(noise)
        else:
            i = np.random.randint(0, self.x_train.shape[0],
samples)
            images = self.x_train[i, :, :, :]

        plt.figure(figsize=(10,10))
        for i in range(images.shape[0]):
```

```
                plt.subplot(4, 4, i+1)
                image = images[i, :, :, :]
                image = np.reshape(image, [self.img_rows,
self.img_cols])
                plt.imshow(image, cmap='gray')
                plt.axis('off')
            plt.tight_layout()
            if save2file:
                plt.savefig(filename)
                plt.close('all')
            else:
                plt.show()
```

使用 MNIST_DCGAN 的 train()方法完成定义所有的主要训练任务。

参数如下。

train_steps：训练步数，默认为 2000 步。

batch_size：批训练大小，默认每批 256 个样本。

save_interval：每隔多少个训练步骤保存生成的 png 效果图，默认不保存训练效果。

另外，有一些变量需要着重解释。noise_input 代表生成模型的随机噪音输入，此变量只用来保存生成的 png 效果图，不作他用。实际的训练从 for 循环开始，即：

```
    for i in range(train_steps):
        ......
```

可见，实际的批次训练由判别模型 discriminator 和对抗模型 adversarial 交替完成，使用.train_on_batch 方法完成实际训练参数更新。每次更新，判别模型的输入为两倍的 batch 尺寸，一半的尺寸是真实输入，一半的尺寸是

生成器的伪造输入，判别器必须以更好地区分这两部分输入为目标。另一方面，对抗模型的输入为 1 个 batch 尺寸的随机噪音数据，输出时对抗模型中的生成器和判别器一同训练，生成器以欺骗判别器为目标更新网络参数，从而达到最终模型真实 MNIST 手写数字集的效果。另外，在每一个指定训练间隔打印 loss 数值，并且保存生成器生成的模拟图片。

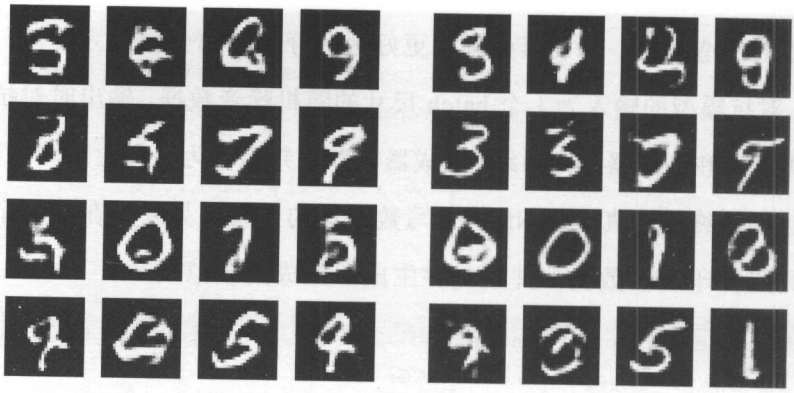
```
    if __name__ == '__main__':
        mnist_dcgan = MNIST_DCGAN()
        timer = ElapsedTimer()
        mnist_dcgan.train(train_steps=10000,   batch_size=256,
save_interval=500)
        timer.elapsed_time()
    mnist_dcgan.plot_images(fake=False, save2file=True)
```

整个程序的入口函数中，我们调用训练任务，以 500 个训练间隔保存训练效果。实际训练结果如图 9-5 所示。

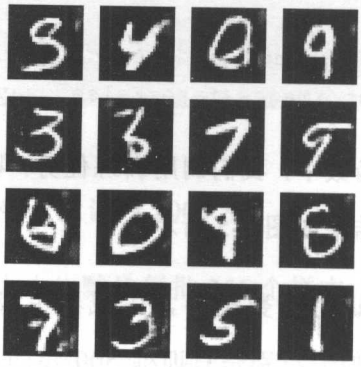
观察后注意到，训练步数越多，生成器生成的伪造样本与实际的 MNIST 手写数字更相似。特别是只训练 500 步的情况下，生成器的伪造数字模糊不清，非常难识别。

如读者希望深入研究，本章代码请详见：

https://github.com/yanchao727/keras_book



500 步训练后　　5000 步训练后



10 000 步训练后

图 9-5　DCGAN 模型在特定训练步数后的生成数据样例

参考文献

[1] https://github.com/fchollet/keras/blob/master/CONTRIBUTING.md

[2] https://github.com/bstriner/keras-adversarial

[3] https://github.com/jacobgil/keras-dcgan

[4] http://torch.ch/blog/2015/07/30/cifar.html

[5] http://groups.csail.mit.edu/vision/TinyImages/

[6] http://cs231n.github.io/convolutional-networks/

[7] Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. In: ICLR (2016)

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets.In NIPS, pages 2672–2680. 2014

[9] http://nooverfit.com/wp/9-%E7%94%9F%E6%88%90%E5%AF%B9%E6%8A%97%E7%BD%91%E7%BB%9C101-%E7%BB%88%E6%9E%81%E5%85%A5%E9%97%A8-%E9%80%9A%E4%BF%97%E8%A7%A3%E6%9E%90/

[10] http://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-generative- models-and-adversarial-training-upc-2016

[11] http://www.leiphone.com/news/201612/Cdcb1X9tm1zsGSWD.html

[12] https://github.com/vdumoulin/conv_arithmetic

[13] https://nlp.stanford.edu/projects/glove/

[14] Pennington J, Socher R, Manning C D. Glove: Global Vectors for Word Representation[C]//EMNLP. 2014, 14: 1532-1543

[15] Schnabel T, Labutov I, Mimno D, et al. Evaluation methods for unsupervised word embeddings[C]//Proc. of EMNLP. 2015

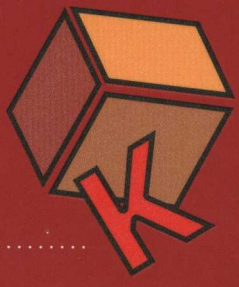
Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

深度学习
Keras快速开发入门

许多读者朋友在学习深度学习时，往往会被复杂的数学公式和理论所困扰，以至于常常无从下手，从 TensoFlow 难以理解的设计和表述方法到 Caffe 冗长的神经网络模型定义，很难清晰、完整和快速地掌握深度学习技术。本书选择 Keras 这一深度学习框架向读者介绍深度学习技术和应用，力求使用简洁、高效和丰富的实例帮助读者快速掌握这门技术，这得益于 Keras 良好的模块化、极简的设计、易扩展性和快速原型迭代等特点。通过学习本书的内容，读者能够快速搭建满足产品需求的神经网络模型，从而加快产品研发周期。

通过本书，您将学到：

- Keras 框架诞生的历史，以及其特点，优势。
- keras 的安装和配置。
- Keras 的网络结构。
- Keras 的数据预处理方法。
- Keras 的调试技巧和可视化工具。
- 如何用 Keras 快速构建深度学习原型并着手实战。



作者简介

乐毅：
计算机专业硕士，现任职于某人工智能企业深度视觉项目，资深深度视觉架构师。负责公司利用深度学习技术在人脸识别的应用和开发，对深度学习及人脸识别技术具有浓厚兴趣。擅长 Caffe、Keras 和 TensoFlow 等深度学习框架的开发和应用。

严超：
计算机专业硕士，现任职于某大型信息科技企业，高级软件工程师。负责公司深度学习，云计算等技术领域的应用研究及相关项目，对机器学习、云计算、数据深度挖掘具有浓厚兴趣。擅长 Keras 等深度学习框架及网络模型应用。

个人博客：http://www.nooverfit.com。

上架建议：人工智能>深度学习

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：孙学瑛
封面设计：吴海燕

ISBN 978-7-121-31868-9

定价：69.00元